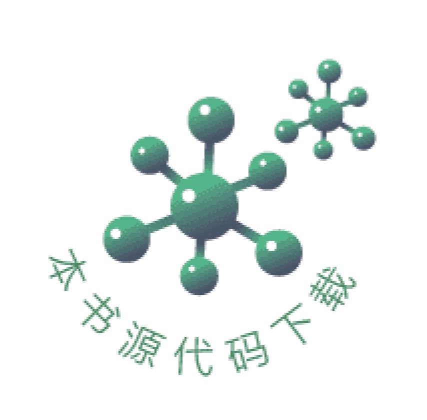
去繁化简地梳理深度学习的理论知识
浅显易懂地讲解算法原理
案例实践教学使你快速学习并掌握常用的深度学习模型



• 杨云 杜飞 著 •

深度学习实战

清華大学出版社



• 杨云 杜飞 著 •

深度学习实战

清華大学出版社
北京

内容简介

深度学习为人工智能带来了巨大突破，也成为机器学习领域一颗闪耀的新星。虽然相关学习资料丰富，但大部分内容较为庞杂且难以理解，并对初学者的相关理论知识与实践能力有较高的要求，这使得大部分想进入这一领域的初学者望而却步。本书去繁化简地对深度学习的理论知识进行了梳理，并对算法实现做出了浅显易懂的讲解，适合初学者进行学习。结合本书的内容，读者可以快速对深度学习进行实践。通过启发式的自学模式，可以使读者由浅入深地学习并掌握常用的深度学习模型，为进一步使用开源的深度学习平台与工具提供理论与实践基础。

本书可作为高等院校计算机专业的本科生或研究生教材，也可供对深度学习感兴趣的研究人员和工程技术人员阅读参考。

图书在版编目（CIP）数据

深度学习实战 / 杨云，杜飞著.—北京：清华大学出版社，2018
ISBN 978-7-302-49102-6

Ⅰ.①深…　Ⅱ.①杨…　②杜…　Ⅲ.①机器学习　Ⅳ.①TP181

中国版本图书馆 CIP 数据核字（2017）第 302059 号

责任编辑：夏毓彦
封面设计：王　翔
责任校对：闫秀华
责任印制：刘祎淼

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者：北京密云胶印厂
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　张：**20.75　　**字　数：**532 千字
版　　次：2018 年 1 月第 1 版　　**印　次：**2018 年 1 月第 1 次印刷
印　　数：1~3000
定　　价：69.00 元

产品编号：075721-01

前　言

随着谷歌的 AlphaGo，IBM 的 watson 以及百度的智能机器人百小度的问世，人工智能成为了大众热烈讨论的焦点，而作为这些智能产品的核心技术，深度学习受到了学界与产业界的广泛关注。深度学习凭借其优良的性能，被广泛应用于计算机视觉、图像分析、语音识别和自然语言处理等诸多领域中。但深度学习的算法与模型较为复杂，对于初学者来说较难理解与掌握，需要其有一定的理论与实践应用基础。本书作者通过把理论知识与大量实践例子相结合，运用易懂与诙谐的语言为初学者呈现了一部指导深度学习实战的首选之作。本书的面向对象为计算机及相关专业的本科生、研究生，以及相关领域的初级研究人员。与同类著作不同的是本书更强调读者的亲身实践，分为模块设计与代码实践两部分，当读者学习完模块设计部分的理论知识后，还可以在实践代码的关键位置添加自己的代码，并测试实现的深度学习模型的每一个关键环节，以此进一步理解与掌握所学的算法与模型。

本书共分为 8 章，第 1 章为深度学习的发展介绍，其他 7 章对深度学习的理论知识和应用进行了深入浅出的讲解，分别为第 2 章 机器学习快速入门，第 3 章 前馈神经网络，第 4 章 深度学习正则化，第 5 章 深度学习优化，第 6 章 卷积神经网络，第 7 章 循环神经网络，第 8 章 TensorFlow 快速入门。每一个章节在其结尾部分都会提出深度学习算法与模型的实践学习，按照作者的设计步骤，读者可以逐步完成代码的编写，并对其进行测试，最终完成整个算法与模型代码的实践。本书不同于传统理论介绍+代码演示书籍之处在于，理论知识与实践学习部分可以分开阅读，其每一章节的实践学习部分更加强调与读者的互动性。本书还精心设计了许多子模块，给予大量的编程提示，并引导读者通过自学的方式完成各个子模块的实现，进而强化读者对不同模块编码实现的学习与理解，在每章末尾都会给出相应的参考代码。

本书的作者特别感谢相关科研项目与人才计划的支持，其中包括：国家自然科学基金项目（61402397，61663046），云南省科技厅应用基础研究计划面上项目（2016FB104），云南省软件工程重点实验室开放基金面上项目（2015SE201），云南大学数据驱动的软件工程省科技创新团队（2017HC012），云南省中青年学术和技术带头人后备人才计划（2017HB005），云南省百名海外高层次人才引进计划。

深度学习相关研究领域的发展日新月异，本书作者自认才疏学浅，只略知其中一二，书中内容的设计与撰写是作者对深度学习的个人认识与理解，由于水平有限，如有不妥之处请广大读者不吝赐教。

本书免费提供了云盘下载文件，内容包括书中所有综合案例的素材文件，下载地址为：**https://pan.baidu.com/s/1nvacrYL**（注意区分英文字母大小写），如果下载有问题，请发送电子邮件至 booksaga@126.com，邮件主题设置为“深度学习实战”。

杨云

2017 年 8 月 29 日于云南大学

目　录

第 1 章

深度学习的发展介绍

一日清晨，朝阳未热，少年小飞未醒，接到了一个陌生的电话，电话中念到“双眸剪秋水，一手弹春风，歌尽琵琶怨，醉来入梦中。”这首诗如何？电话中是一位声音甜美，自称小鱼的女生，听到这，昏沉朦胧的少年，突然振作了，既疑惑又激动，思考一会儿，紧张又兴奋地说道“虽然我不懂诗，但感觉剪和弹用得非常妙呀。请问我认识你吗？”“哎呀，不好意思，我打错电话啦，我还以为你是我闺蜜”，电话那头娇羞地答到。机智如小飞，肯定不会错过这美丽的错误，然后就立即回道……以上故事，是我将图灵在 1950 年《机器与智能》中关于“模仿游戏”[1]的一段改编，而上述诗句是来自于中科院院士张钹展示的机器人所做的诗。

随着现代技术的进步，特别是**深度学习**（Deep Learning）[2]的发展，完成上述任务并不再是天方夜谭。我们不妨和图灵一起思考，如果机器能够完成上述的问答，那小飞怎么去判断是人或者是机器呢？那所谓的意识又是什么呢？带着这些令人头痛但又激动的思考，我们现在就开始深度学习的入门之旅。

在一百多年前，可编程计算机第一次被构想出时，人们就想象着机器是否可以变得智能。如今，人工智能已是一个欣欣向荣的领域，拥有着许多实际应用及激动人心的研究主题。我们期望智能软件能够应用到自动化家居、语音识别、图像理解、医疗诊断及辅助基本科学的研究中去。

人与计算机就像天平的两端，机器善于计算，只要问题能被一系列数学规则形式化描述出来，人们就可以编写程序让机器执行。比如求解线性方程组，计算行星轨迹等，虽然这些

任务很复杂，但对于机器而言却很简单。但有些问题虽然看似简单，但人却难以描述，比如人如何识别口语，文字或脸部图像。由于人难以描述这些本能就会的任务，虽然人处理很简单，但对于机器而言却难于上青天。而人工智能的目标就是去解决人容易执行，但很难形式化描述的任务。

在早期的人工智能中，我们想要机器智能地完成某项任务，我们首先需要使用形式化语言，硬编码关于该任务的知识；然后计算机通过这些形式化语言，自动使用逻辑推理规则去推理状态。但其中根本问题在于，我们要知道如何解决某项问题。比如，我们想教机器人下棋，我们知道确定的规则，也知道某些下棋的技巧，然后我们将各种规则、各种技巧通过编程实现出来，那么机器就学会了下棋。机器棋艺的高低，其实只是编程人员棋艺的高低，机器只是比人类运算快一些罢了。但有些事情，我们是很难描述清楚的，我喜欢一朵花，一首诗，一个人，但问我为什么喜欢，我却不知道。同样地，你想让机器去赏析一首诗，一朵花，一个人，那也是很难完成的任务。

深度学习便是这些更直觉问题的一种解决方案。这种解决方案允许电脑**从经验中学习**并依靠**层次化概念**去理解世界。通过从经验中收集知识，这种方法避免了人类手工地去形式化列举电脑所需的知识，层次化概念允许电脑从更简单的概念中学习更抽象的概念。也许现在你已经有点晕眩了，那我们就先轻松地讲一个故事吧。

认识小鱼同学后，为了博取小鱼同学的欢心，小飞就想亲手做一道黄焖鸡给小鱼。于是他就翻看了一下食谱，开始了“实验”：首先，准备各种食材；然后，将各种配菜预炒，再放入鸡块爆炒；爆炒入味后再放入一碗水，用锅盖焖 10 分钟，最终黄焖鸡就出锅了。小飞尝了尝亲手做的黄焖鸡，如你所想，才将鸡块入口，就吐了出来，因为实在太难吃了。但小飞并没有灰心，总结了一下，少放了点水，多加了点盐，再多炒了会儿鸡块，鼓起勇气一尝。好咸啊！虽然又失败了，但总体还是挺开心的，因为至少可以吃了。一鼓作气，再次总结，再次实验，忐忑地再次试尝，总算有了黄焖鸡该有的味儿了。起锅打包，小飞开心地去找小鱼同学去了……

从以上的故事中，我们发现了什么？发现小飞买了很多鸡肉，我们知道了，小飞一开始不会做黄焖鸡，但通过不断尝试，是可以做好黄焖鸡的。在此期间，小飞做了什么事呢？其实只是在调整，放多少水，放多少盐，煮多久，炒多久等，而这个过程就是一个简单的学习行为。同理，在机器学习中，我们并不是将做黄焖鸡的整个过程，每个细节都编程给机器，相反，我们把小飞的学习过程，复制给机器。做一次黄焖鸡，我们可以将其称为一个**数据**，而放水，放盐，煮多久，我们可以称为数据的**特征**（Feature），对于学习最朴素的理解其实就是调整数据特征各自的重要性。

我们再仔细剖析一下上面的故事，有一个过程叫作食材准备阶段，我们可以简单地将该阶段作为“黄焖鸡”数据的一个特征，当然，我们也可以将这一特征再细分一下，多少姜、多少辣椒配多少鸡块呢？而酱油的多少和水的多少又如何调配呢？那这些又如何影响爆炒和黄焖的时间呢？数据的特征直接影响着学习的难度及最终结果的好坏。如果我们先学习油盐酱醋的调配，将其组成配料特征，这样就简化了整个学习过程，而这种特征到特征的学习，我们就称为**表征学习或表示学习**（Representation Learning[3]，“表征”是一个心理学词汇，翻译成表征更贴切些，但“表示”更常用些）。如果我们将做黄焖鸡的过程,细分得非常细致，也就是**特征维度**（数量）很高，那么每一维度对最终学习任务好坏的影响就越小，如香菇的

量对黄焖鸡的好坏影响很小，我们很难通过调整香菇的量，去调整黄焖鸡的做法，而香菇特征也相互影响着其他特征的选择。因此我们先学习配菜这一简单的概念或特征，然后再学习一些更抽象的概念，一层一层的抽象，最终仅仅去学习“选材”“黄焖”这些特别抽象的特征，这就是所谓的**深度学习**。

1.1　如何阅读本书

本书是一本有关深度学习的入门实战教程，目的在于尽可能以一种轻松的方式，讲解一些深度学习核心的技术，关键的思想，以及常用的技巧方法。需要注意的是，深度学习充斥着大量的数学公式，大多数人可能会望而生畏，看着公式就头痛欲裂。但数学只是工具，数学公式只是简化我们的描述，我们更应该做的是深入理解这些公式背后的哲学内涵。本书同样会列出很多的公式，但请读者们不要在意所谓的公式，应该多去看一些文字性描述，理顺公式的每一次演变，公式是帮助你快速的记忆，并不是你的负担。也许，对于大多数读者，看懂各种算法，了解各种思想，但编写程序还是无从下手，本书将使用 IPython Notebook 进行模块化编程练习，我们会一步一步地动手实践，希望能帮助你从理论走向实践，并从实践中加深对理论知识的进一步理解。

深度学习能够火爆的最主要原因是大数据的到来，以及运算能力的大大提高，针对 GPU 编程的 Theano，TensorFlow，Torch 等深度学习库，都是非常好的学习资源，如果你想从事深度学习研究，则应该至少掌握一种以上的深度学习库。但本书的目的在于让你不那么头痛欲裂地跨入深度学习领域，本书并不是一本深度学习平台指导用书。在本书的最后一章，我们会教你如何搭建 TensorFlow 深度学习库，帮助你铺垫更广阔的知识世界。我们希望能给你一个小板凳，然后你可以站上去，希望你拥有瞭望远方的喜悦，希望你能享受在微风中的呼吸。因此，如果你有些惧怕数学公式或编程能力相对较弱，这些都没关系。最重要的是你憧憬着深度学习，相信集体的力量，渴望着人工智能，同时也不太在意一些稍显不严谨的语言，那或许这是一本属于你的书。需要注意的是，本书作为入门书籍，只会重点介绍深度学习在实际应用中的技术及方法，深度学习的一些高级研究主题，如**自动编码器**（Autoencoder）[4]，**受限玻尔兹曼机**（RBM）[5]等非监督学习研究主题并不涉及。但这些主题是深度学习的研究重点，对于有志从事深度学习研究的专业人员而言，这些主题才是更广阔的世界。

每章我们都会分为两个部分，第一部分介绍该章节的深度学习技术，第二部分进行编程练习，在每章的末尾，我们给出了参考代码，希望在你想要放弃时给你点帮助。本书的知识内容主要参考于 Ian Goodfellow，Yoshua Bengio，Aaron Courville 所著的 *Deep Learning* 以及斯坦福大学的 CS231 公开课，你也可以将本书作为学习这些书的铺垫。

1.2　深度学习沉浮史

喜欢一个人，就应该去了解她的过去，感受她的曾经。想要学习深度学习，不妨也粗略地了解下它的过去，下面我们列出了一些深度学习的关键趋势。

- 深度学习拥有悠久的历史，曾经几度“改嫁”，反映着不同的哲学观点，并且流行度也是几度兴衰。

- 深度学习随着可训练数据的不断增长变得更加有用。
- 随着计算机硬件，软件基础设施的提高，深度学习模型尺寸正在变大。
- 随着精确度不断地提高，深度学习已经开始应用于越来越复杂的领域。

“滚滚长江东逝水，浪花淘尽英雄，是非成败转头空，青山依旧在，几度夕阳红……”许多读者或许已经听说过深度学习作为一种令人激动的新技术，刺激着整个人工智能领域的发展，并且让整个社会前所未有地讨论着人工智能这一话题。但事实上，深度学习最早可以追溯到二十世纪四十年代。深度学习之所以似乎是新的，是因为其流行度几经沉浮，说得严重些，深度学习以前是遭到同行鄙视的。深度学习历经多次“良妻改嫁”并跟随“夫君”改名多次，并且只是最近才称为“深度学习”。其实这个领域已经更名了许多次，其也映射出不同学者，不同思想对该领域的影响。

泛泛地说，深度学习的发展有三段起伏：在 1940—1960 年，深度学习被称为**控制论**，随着生物学习理论的发展，以及感知机模型的实现，爆发了第一波热潮。第二波浪潮为 1980—1995 年，开始于**联结主义**[6]方法，随着使用**反向传播算法**训练 1~2 个隐藏层的神经网络而迅速“膨胀”。而当前以深度学习之名复苏是始于 2006 年，由于使用**逐层贪婪非监督式预训练**[4]方法初始化深层神经网络权重，使得网络在数据集较少时依然可以有效地训练深层的神经网络，而在 2012 年后深度学习彻底被工业界所接纳，大量的公司及科研人员开始关注了深度学习。

1.2.1 模拟生物大脑的疯狂远古时代

大浪淘沙，如今洗尽铅华的一些算法是模拟生物学习的计算模型。深度学习过去的名字也被称为**人工神经网络**（ANNs）[7]。深度学习模型也可以说是受生物大脑（无论人类大脑还是动物大脑）启发的工程系统。并且，用于机器学习的神经网络有时也用于帮助理解大脑的功能。神经网络的思考方式主要基于两种观点，一种观点是，大脑证明了智能行为是可能的，那么去构建智能最直接的方式就是模拟大脑，复制其功能；另一种观点则相反，期望通过神经网络去理解人类智慧及其原则。因此，神经网络模型除了解决工程问题外，还试图解释人类智慧等基本科学问题。

现代的术语“深度学习”已经超出了神经科学在当前机器学习模型上的观点，其引出了一个更通用的学习原则，即**多层次组合的原则**，这个原则能够应用于不拘泥于神经元启发的机器学习框架。

现代深度学习的鼻祖是受神经科学观点启发的简单线性模型。该模型如上述的“黄焖鸡模型”一样，使用一组 n 维输入值 x_1，…，x_n 作为数据，然后将它们与输出 y 关联起来。该模型能够学习一组权重 w_1，…，w_n 并且计算它们的输出。

神经网络研究的第一个高潮以**控制论**著称。麦卡洛克－皮茨神经元[8]是一种早期的脑功能模型，该线性模型通过测试 $f(x, w)$ 值的正负，能够识别两种不同的输入分类。当然，为了该模型符合期望的分类定义，其权重需要正确设置，但这些权重需要通过**人类操作设置**。在 20 世纪 50 年代，感知机[9]成为第一个从给定的输入样例中学习分类权重的模型。大约在相同时期，**自适应线性单元**（ADALINE）[10]同样能够自动地学习预测数字数据。这些成就虽然很简单，但给了当时的人工智能领域很大的冲击。因为当时的主流思想是计算机能够做正

确的**逻辑推理**将本质上解决人工智能问题。一时间神经网络受到了大量的关注与投资，但其中也有许多不切实际的想法，因此也受到了许多学者的“仇视”。MIT 人工智能实验室创始人 Marvin Minsky 和 Seymour Papert 就是持怀疑态度的两位学者。1969 年，他们在一本开创性著作中表达了这种质疑，书中严谨分析了感知机的局限性，书名很贴切，就叫《感知机》，其理论证明了感知机无法学习异或函数。看到线性模型中这些缺陷的批评者们，掀起了强烈反对生物启发学习的浪潮，这也导致了神经网络的第一次主要衰败。

但这些简单的学习算法深远地影响了现代机器学习的格局。用于学习自适应线性单元权重的训练算法是随机梯度下降算法的特例，而随机梯度下降算法的稍微修改版本仍然是如今深度学习领域统治级的训练算法。

如今，对于深度学习研究而言，神经科学被当作是一种重要的灵感源泉，但其不再是该领域主要的指导了。而神经科学在深度学习领域中被削弱的主要原因在于我们对大脑还没有足够的认知，并使用其作为指导。

神经科学给了我们理由去期望，一个单独的深度学习算法能够解决许多不同的任务。神经科学家已经发现，如果雪貂的大脑被重新连接，将视觉信号传送到听觉处理区域，雪貂就能通过它们大脑的听觉处理区域学习“看”。我们也许可以假设，大多数哺乳动物大脑可能使用一个单独的算法去解决大脑中的大多数不同任务。在这假设之前，机器学习研究是碎片化的，不同的研究社区研究自然语言处理、视觉、运动以及语音识别任务。如今，这些应用领域仍然分离，但对于深度学习研究团体来说，同时研究许多甚至是全部应用领域是常见的。

我们能够从神经科学中获得一些粗糙的指导方针，如**神经认知机**[11]受到哺乳动物视觉系统的启发，引入了一种强大的图像处理模型结构，并在之后成为**卷积神经网络**[12]的基础。虽然神经科学是深度学习重要的灵感来源，但并不需要将其作为一种刚性指南，真实神经元相比于神经元模型来说复杂得多，但使用更真实的神经元并没有提高机器学习的性能。同时，虽然神经科学已经成功地启发了一些神经网络结构，但由于我们对于生物学习还没有足够多的了解，神经科学还无法给我们训练这些结构提供太多的指导。

各种宣传经常强调深度学习与大脑的相似性。虽然深度学习研究相比于其他的机器学习研究，如核函数机或贝叶斯统计，确实更像去模拟大脑工作，但并不应该仅仅把深度学习作为模拟大脑的一种尝试。现代深度学习灵感来源于许多领域，尤其是应用数学基础，像是线性代数，概率论，信息论及数值优化等。虽然有一些深度学习研究以神经科学作为重要的灵感来源，但其他一些完全和神经科学无关。

需要注意的是，在算法级别努力理解大脑如何工作依然是盛行的，这种努力主要以“计算神经学”著称，并且是深度学习的一个单独的研究领域，研究人员反复地在这两个领域移动是很平常的。深度学习领域主要关心**怎样构建计算机系统，来成功地解决所需的智能任务；**计算神经学领域主要关心**构建更精确模型，模拟大脑如何真实工作**。

1.2.2　联结主义近代

在 20 世纪 80 年代，出现了第二波神经网络研究高潮，并被称为联结主义或并行分布处理 [3]。认知科学是以一种跨学科的方法组合多个不同水平的分析去理解思维，而联结主义就出现在认知科学的这一背景下。在 20 世纪 80 年代早期，大多数认知科学家研究符号推理模

型，尽管当时非常流行，但符号模型非常难于解释大脑如何能够真实地使用神经元执行这些符号。因此联结主义开始研究能够依据神经元执行的**认知模型**[13]。联结主义的核心思想是大量简单的计算单元连接在一起能够完成智能行为，这种观点同样适用于生物神经系统的神经元及计算模型的隐藏单元。并且在 20 世纪 80 年代随着联结主义兴起的一些关键概念，仍然是如今深度学习的核心。

其中之一是**分布式表征**[14]。这种观点是系统的高级或抽象特征由多个子特征组合而来，并且通过组合不同的子特征，系统可以构造不同的高级特征。例如，假设需要有一个视觉系统能够识别汽车、卡车和鸟，并且这些对象可能是红色、绿色或蓝色。一种表述方式是使用独立神经元：红卡车、红汽车、红鸟、绿卡车等。这需要 9 个不同的神经元，并且每个神经元必须独立地学习颜色或对象概念。另一种提高的方式是使用分布式表征，三个神经元描述颜色，三个神经元描述对象。这仅仅需要 6 个神经元而不是 9 个，并且描述红色的神经元能够学习汽车，卡车以及鸟类图像中的红色，不仅仅局限于特定对象的分类图片。

联结主义的另一个成就是成功地使用**反向传播算法**[3]训练深度神经网络。该算法的流行度也是跌宕起伏，但如今依然是训练深度模型统治级的方法。

神经网络研究的第二个高潮持续到 20 世纪 90 年代中期，当基于神经网络和其他 AI 技术的投资者寻找投资时，又开始表现不切实际的雄心壮志，当智能研究无法满足这些不合理的期望时，投资者非常的失望。同时，其他机器学习模型也在飞速地发展，如核函数机[15]，概率图模型[16]都在许多重要任务中取得了良好的结果。这两个因素导致了神经网络流行度的衰退，这种衰退一直持续到了 2007 年。

虽然处于低迷期，但一些人仍然没有将其放弃。加拿大高级研究所（CIFAR）通过其神经计算与自适应感知机（NCAP）研究计划保持神经网络研究的存活。该项目联合了由多伦多大学的 Geoffrey Hinton，蒙特利尔大学的 Yoshua Bengio 以及纽约大学的 Yann LeCun 领导的机器学习研究团队，并且还囊括了一大批顶尖的神经科学，人类和计算机视觉方面专家。

1.2.3 百花齐放，层次结构主导，模型巨大的当代

神经网络研究的第三波高潮开始于 2006 年的重大突破。Geoffrey Hinton 展示了一种名叫**深度置信网络**（Deep Belief Networks，DBN）[17]的神经网络，它通过使用逐层贪婪预训练的方式能够高效地训练网络。其他的 CIFAR 附属研究机构也迅速地展示了同样的策略可以应用在许多其他的深度网络中，并且系统地提高了测试样例的泛化能力。这波神经网络的研究，通俗化地使用深度学习术语来强调深层概念的重要性。相比之前，研究者能够训练更深的神经网络，并且关注深度理论。这时，深度神经网络性能远远超过了基于其他机器学习技术的智能系统和手工设计的功能。时间到了 2012 年，Hinton 领导的深度学习小组使用 ALexNet[18]网络，在 ImageNet 计算机视觉挑战赛上，将当时最佳图像识别性能提高了一倍。从此引爆了学术界及工业界，并且 ALexNet 网络和传统的 CNN 网络并没有本质区别，其核心只是选用了更简单的 ReLU 神经元并使用 GPU 进行加速学习。人们开始重新审视一大批以前遭到抛弃的深度学习模型，并且各大公司也开始了超大规模深度模型的研究，超大规模数据集的“军备竞赛”也正在上演。

如今，神经网络的第三个高潮依然持续着，但深度学习研究的焦点在这波高潮期间已经

戏剧化地改变了，第三波高潮开始关注新非监督学习技术以及深度模型在小数据集的泛化能力。但这些高级主题还处在实验室研究之中，由于篇幅有限，本文仅介绍被成熟应用于工业界的监督式学习算法，有兴趣的读者可以参阅 Ian Goodfellow 所著的 *Deep Learning* 的高级研究主题。

总之，深度学习是机器学习的一个重要分支，并且随着过去几十年的发展，已经吸收了大量神经科学，统计学以及应用数学的知识。在最近几年中，由于计算机更强的能力，更大的数据集以及一些训练深度网络的新技术兴起，其流行度已经获得了极大的增长。但正如这些年神经网络的跌宕起伏一样，目前深度学习虽然处于上升期，但有可能会来到一个梦想破碎的蛰伏期，没有人能说得准，甚至还有大量的科学家开始担忧起人工智能可能会是一个潘多拉魔盒。

但勇敢不是不畏惧，而是心怀恐惧，仍然向前。我们使用图灵于 1950 年《机器与智能》论文中的最后一句话，作为我们的开始，接下来，我们就将正式进入本书的学习。

We can only see a short distance ahead，but we can see plenty there that needs to be done.

——Alan Mathison Turing

1.3　Python 简易教程

Python 是一种非常简单易学的解释性语言。由于强大的开源库支持（NumPy、Scipy、Matplotlib），其广泛应用于科学计算中。如果你励志成为一名数据科学家或数据“攻城狮”，那么 Python 就是你必须要学会的工具之一。接下来我们将简短地介绍下 Python、NumPy、Matplotlib 的使用，如果你已经十分熟悉这些内容，可以轻松地跳过该章节，直接进入下一章节的学习。本章节教程内容主要参考于斯坦福大学 cs228 课程的 Python 教程，具体详情可以使用下列网址查看：

https://github.com/kuleshov/cs228-material/blob/master/tutorials/python/cs228-python-tutorial.ipynb。

1.3.1　Anaconda 搭建

Anaconda 是开发 Python 最常用的开源平台之一，已经为你安装了 Python 中最常用的工具库（NumPy、Matplotlib、Scipy、IPython 等），使用起来非常方便，读者可以访问 https://www.continuum.io/downloads/网址进行下载。本书的教程将使用 Python2.7+版本，因此确保你下载的版本符合我们的要求。在本节中，将逐步学习以下内容。

- Python 基本使用：基本数据类型（Containers、Lists、Dictionaries、Sets、Tuples），函数，类；
- NumPy：数组，数组索引，数据类型，数组运算，广播；
- Matplotlib：Plotting，Subplots，Images。

1.3.2　IPython Notebook 使用

IPython Notebook（现改名 Jupyter Notebook）是一种基于 Web 技术的交互式计算文档，

使用浏览器作为客户端进行交互，其页面被保存为.ipynb 的类 JSON 文件格式，是非常高效的教学演示工具。本书的教学教程主要就使用 IPython Notebook 进行分模块演示。安装 Anaconda 时，默认就安装了 IPython Notebook，读者只需直接启动即可。

- *启动* IPython Notebook

首先启动 Jupyter：如图 1-1 所示，启动 dos 窗口，将路径转换到文件："第 1 章练习-numpy.ipynb"所在路径，然后输入 jupyter notebook（或 ipython notebook）命令，再按 Enter 键确认。这时，浏览器会自动启动 Jupyer。

```
管理员: 命令提示符 - jupyter notebook
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation。保留所有权利。

C:\WINDOWS\system32>cd f:\DLAction

C:\WINDOWS\system32>f:

f:\DLAction>jupyter notebook
[I 19:16:23.976 NotebookApp] [nb_conda_kernels] enabled, 2 kernels found
[I 19:16:28.764 NotebookApp] [nb_anacondacloud] enabled
[I 19:16:29.379 NotebookApp] \u2713 nbpresent HTML export ENABLED
[W 19:16:29.379 NotebookApp] \u2717 nbpresent PDF export DISABLED: No module named 'nbbrowserpdf'
[I 19:16:29.464 NotebookApp] [nb_conda] enabled
[I 19:16:29.701 NotebookApp] Serving notebooks from local directory: f:\DLAction
[I 19:16:29.701 NotebookApp] 0 active kernels
[I 19:16:29.701 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
[I 19:16:29.701 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
```

图 1-1 启动 Jupyter Notebook 命令行示意图

1.3.3 Python 基本用法

Python 是一种面向对象的解释型高级编程语言。很多时候，由于其代码具有高可读性，且只需要数行代码就可以表达复杂的功能，使 Python 看起来简直和伪代码一样。如下列代码所示，为 Python 实现经典的快速排序算法例子。

```
def quicksort( arr ) :
    if len(arr) <= 1:
        return arr
    pivot = arr[ len( arr ) / 2 ]
    left = [ x for x in arr if x < pivot ]
    middle = [ x for x in arr if x == pivot ]
    right = [ x for x in arr if x > pivot ]
    return quicksort( left ) + middle + quicksort( right )
print quicksort( [ 3,6,8,10,1,2,1 ] )
```

```
[1, 1, 2, 3, 6, 8, 10]
```

1.3.3.1 基本数据类型

和大多数编程语言一样，Python 拥有一系列的基本数据类型，比如整型、浮点型、布尔型和字符串等。这些基本数据类型的使用方式和其他语言的使用方式类似。

- 整型和浮点型

输入：	输出：
x = 3 print x, type(x)	3 <type 'int'>
print x + 1　　# 加；	4
print x - 1　　# 减；	2
print x * 2　　# 乘；	6
print x ** 2　# 幂乘；	9
x + = 1 print x　# 打印 "4"。	4
x *= 2 print x　# 打印 "8"。	8
y = 2.5 print type(y) # 打印 "<type 'float'>"。 print y, y + 1, y * 2, y ** 2 # 打印 "2.5 3.5 5.0 6.25"。	<type 'float'> 2.5 3.5 5.0 6.25
# python 不支持 (x++) 或(x--) 运算。 print y++	File "<ipython-input-6-a30cd8b33996>", line 2 print y++ ^ SyntaxError: invalid syntax

- 布尔型

Python 实现了所有的布尔逻辑，但使用的是英语单词（and、or、not 和 xor），而不是我们习惯的操作符（&&和||等）。

输入：	输出：
t, f = True, False print type(t) # 打印 "<type 'bool'>"。	<type 'bool'>
print t and f　# 逻辑 AND；	False
print t or f　　# 逻辑 OR；	True
print not t　　# 逻辑 NOT；	False
print t != f　　# 逻辑 XOR；	True

- 字符串

输入：	输出：
hello = 'hello'　　# 字符串可以使用单引号。 world = "world"　　# 也可以使用双引号。 print hello, len(hello),world	hello 5 world

hw = hello + ' ' + world　# 字符串拼接。 print hw　# 打印 "hello world"。	hello world
hw12 = '%s %s %d' % (hello, world, 12)　# 按格式输出。 print hw12　# 打印 "hello world 12"。	hello world 12

也可以将字符串当作是一个对象，有很多的方法，如下列代码所示。

输入：	输出：
s = "hello" print s.capitalize( )　# 将字符串首字母大写；打印 "Hello"。	Hello
print s.upper( )　　# 将字符串转换成大写；打印 "HELLO"。	HELLO
print s.rjust(7)　　# 字符串向右对齐，使用空格进行占位；打印 "　hello"。	hello
print s.center(7) # 字符串居中，使用空格对左右进行占位；打印" hello"。	hello
print s.replace('l', '(ell)') # 使用子串代替规定处字符。 　　　　　　　　　　# 打印 "he(ell)(ell)o"。	he(ell)(ell)o
print '　wo　rld '.strip( ) # 删除空白字符（开头或结尾）；打印 "wor　ld"。	wo　rld

如果想掌握更多关于字符串的应用与操作，有兴趣的读者可以访问以下网址来进行学习：https://docs.python.org/2/library/stdtypes.html#string-methods。

Python 有 4 种容器类型：列表（Lists）、字典（Dictionaries）、集合（Sets）和元组（Tuples）。

1.3.3.2　列表（Lists）

在 Python 中，列表相当于数组，但是列表长度可变，且能包含不同类型元素。

输入：	输出：
xs = [3, 1, 2]　# 创建列表。 print xs, xs[2]	[3, 1, 2] 2
print xs[-1] # 负值索引相当于从列表的末端进行反向索引；打印 "2"。	2
xs[2] = 'foo'　　# 列表可以包含不同类型的元素。 print xs	[3, 1, 'foo']
xs.append('bar')　# 添加新元素到列表末端。 print xs	[3, 1, 'foo', 'bar']
x = xs.pop( )　　# 移除列表末端元素。 print x, xs	bar [3, 1, 'foo']

如果想掌握更多关于列表的应用与操作，有兴趣的读者可以访问以下网址来进行学习：https://docs.python.org/2/tutorial/datastructures.html#more-on-lists。

- 切片（Slicing）

为了同时获取列表中的多个元素，Python 提供了一种简洁的语法去访问子列表，这就

是切片。

输入：	输出：
nums = range(5) # range 是内置的创建整型列表函数。 print nums　　# 打印 "[0, 1, 2, 3, 4]"。	[0, 1, 2, 3, 4]
print nums[2 : 4] # 获取索引 2-4（排除）的子列表；打印 "[2, 3]"。	[2, 3]
print nums[2 :]　# 获取索引 2 到末的子列表；打印 "[2, 3, 4]"。	[2, 3, 4]
print nums[: 2] # 获取索引开始到 2（排除）的子列表；打印"[0, 1]"。	[0, 1]
print nums[:]　# 获取整个列表；打印 "[0, 1, 2, 3, 4]"。	[0, 1, 2, 3, 4]
print nums[: -1]　# 切片也可以使用负号索引；"打印 [0, 1, 2, 3]"。	[0, 1, 2, 3]
nums[2:4] = ['s',' we'] # 用新子列表替换指定索引列表中的子列表。 print nums　　# 打印 "[0, 1, s, we, 4]"。	[0, 1, 's', 'we', 4]

- 循环（Loops）

可以按以下方式遍历列表中的每一个元素。

输入：	输出：
animals = ['cat', 'dog', 'monkey'] for animal in animals: 　　print animal	cat dog monkey

如果想要在循环体内访问每个元素的指针，可以使用内置的枚举（enumerate）函数，注意：起始 ID 为 0。

输入：	输出：
animals = ['cat', 'dog', 'monkey'] for idx, animal in enumerate(animals): 　　print '#%d: %s' % (idx + 1, animal)	#1: cat #2: dog #3: monkey

- 列表解析（List Comprehensions）

在编程的时候，我们常常要将列表中的每一元素使用特定的表达式进行转换。下面是一个简单例子，将列表中的每个元素转换成它的平方。

输入：	输出：
nums = [0, 1, 2, 3, 4] squares = [ ] for x in nums: 　　squares.append(x ** 2) print squares	[0, 1, 4, 9, 16]

也可以使用更简单的列表解析（List Comprehension）。

输入：	输出：
nums = [0, 1, 2, 3, 4] squares = [x ** 2 for x in nums] print squares	[0, 1, 4, 9, 16]

列表解析也可以包含条件语句。

输入：	输出：
nums = [0, 1, 2, 3, 4] even_squares = [x ** 2 for x in nums if x % 2 == 0] print even_squares	[0, 4, 16]

1.3.3.3 字典（Dictionaries）

字典用来存储（键，值）对，其和 Java 中的 Map 差不多。

输入：	输出：
d = { 'cat': 'cute', 'dog': 'furry' } # 为数据创建字典。 print d['cat'] # 从字典中获取词条(entry)；打印 "cute"。	cute
print 'cat' in d # 检查字典中是否有给定键值(key)；打印 "True"。	True
d['fish'] = 'wet' # 给定键值，创建词条。 print d['fish'] # 打印 "wet"。	wet
print d.get('monkey', 'N/A') # 获取字典中元素默认值；打印 "N/A"。	N/A
print d.get('fish', 'N/A') # 获取字典中元素默认值；打印 "wet"。	Wet
del d['fish'] # 从字典中移除元素。 print d.get('fish', 'N/A') # "fish"不再是字典中的键值；打印 "N/A"。	N/A

字典的详细用法可以访问 https://docs.python.org/2/library/stdtypes.html#dict 网址来进行学习。

- 迭代字典

输入：	输出：
d = { '人': 2, '猫': 4, '蜘蛛': 8 } for animal in d: legs = d[animal] print ' %s 有 %d 腿' % (animal, legs)	猫有 4 腿 蜘蛛有 8 腿 人有 2 腿

也可以使用 iteritems 方法进行迭代。

输入：	输出：
d = { '人': 2, '猫': 4, '蜘蛛': 8 } for animal, legs in d.iteritems( ): print ' %s 有 %d 腿' % (animal, legs)	猫有 4 腿 蜘蛛有 8 腿 人有 2 腿

- 字典解析（Dictionary Comprehensions）

和列表解析类似，字典解析允许你轻松地构造字典，如下列代码所示。

输入：	输出：
nums = [0, 1, 2, 3, 4] even_num_to_square = { x: x ** 2 for x in nums if x % 2 == 0 } print even_num_to_square	{ 0: 0, 2: 4, 4: 16 }

1.3.3.4　集合（Sets）

集合存放着无序的不同元素，在 Python 中，集合使用花括号表示，如果将一个序列转换为集合，那么该序列的重复元素将会被剔除，并且原有的顺序也将被打散。

输入：	输出：
animals = { 'cat', 'dog' } print 'cat' in animals　# 检查是否元素在集合中；打印 "True"。	True
print 'fish' in animals　# 打印 "False"。	False
animals.add('fish')　# 向集合中添加元素。 print 'fish' in animals	True
print len(animals)　# 集合中的元素个数;	3
animals.add('cat') # 添加一个存在的元素到集合中，其没有变化。 print len(animals)	3
animals.remove('cat')　# 从集合中移除一个元素。 print len(animals)	2

- 集合循环

虽然集合中的循环语法和列表中的一样，但由于集合是无序的，因此访问集合元素的时候，不能做关于顺序的假设。

输入：	输出：
animals = { 'cat', 'dog', 'fish' } for idx, animal in enumerate(animals): print '#%d: %s' % (idx + 1, animal)	#1: fish #2: dog #3: cat

- 集合解析（Set Comprehensions）

和字典、列表一样，可以很方便地使用集合解析构建集合。

输入：	输出：
from math import sqrt print { int(sqrt(x)) for x in range(30) }	set([0, 1, 2, 3, 4, 5])

1.3.3.5 元组（Tuples）

元组是一个（不可改变）有序列表。元组和列表在很多方面都很相似，最大的区别在于元组可以像字典一样使用键/值对，并且还可以作为集合中的元素，而列表不行。如下列代码所示。

输入：	输出：

# 通过元组键值创建字典。 	
d = { (x, x + 1): x for x in range(10) }
t = (5, 6) # 创建元组。
print type(t)
	<type 'tuple'>

print d[t]	5
print d[(1, 2)]	1

1.3.3.6 函数（Functions）

Python 使用关键词 def 来定义函数，如下列代码所示。

输入：	输出：

def sign(x):
    if x > 0:
        return '正'
    elif x < 0:
        return '负'
    else:
        return '零'
for x in [-1, 0, 1]:
    print sign(x)
	负
	零
	正

也经常使用可选参数来定义函数，如下列代码所示。

输入：	输出：

def hello(name, loud = False):
    if loud:
        print 'HELLO, %s' % name.upper( )
    else:
        print 'Hello, %s!' % name
hello('Bob')
hello('Fred', loud = True)
	Hello, Bob!
	HELLO, FRED

1.3.3.7　类（Classes）

在 Python 中，定义类的语法很直接，如下列代码所示。

输入：	输出：

class Greeter:
    # 构造函数。
    def __init__(self, name):
        self.name = name　# 创建一个变量实例。
    # 实例方法。
    def greet(self, loud = False):
        if loud:
            print 'HELLO, %s!' % self.name.upper( )
        else:
            print 'Hello, %s' % self.name
g = Greeter('Fred')　# 创建一个 Greeter 类实例。
g.greet( )　　　　　# 调用实例方法；打印 "Hello, Fred"。
g.greet(loud = True) # 调用实例方法；打印 "HELLO, FRED!"。
	Hello, Fred
	HELLO, FRED!

1.3.4　NumPy

NumPy 是 Python 中用于科学计算的核心库，其提供了高性能的多维数组对象及相关工具，其用法和 MATLAB 比较相似，具体详情可以参考如下网址：

http://wiki.scipy.org/NumPy_for_Matlab_Users。

要使用 NumPy，首先要导入 numpy 包：import numpy as np。

1.3.4.1　数组（Arrays）

NumPy 中的数组是由相同数据类型组成的网格，可以通过非负整型的元组进行访问，数组维度数量也被称为数组的**秩或阶**（rank），数组的**形状**（shape）是一个由整数构成的元组，描述数组不同维度上的大小。我们可以从 Python 内嵌的列表中创建数组，然后利用方括号访问其中的元素，如下列代码所示。

输入：	输出：
a = np.array([1, 2, 3]) # 创建秩为 1 的数组。 print type(a), a.shape, a[0], a[1], a[2]	<type 'numpy.ndarray'> (3L,) 1 2 3
a[0] = 5　# 改变数组元素。 print a	[5 2 3]

# 创建秩为 2 的数组。 	
b = np.array([[1, 2, 3], [4, 5, 6]])
print b
	[[1 2 3]
	[4 5 6]]

print b.shape	(2L, 3L)
print b[0, 0], b[0, 1], b[1, 0]	1 2 4

NumPy 同样提供了大量的方法创建数组，如下列代码所示。

输入：	输出：

# 创建 2*2 的零矩阵。 	
a = np.zeros((2, 2))
print a
	[[0.　0.]
	[0.　0.]]

# 创建各元素值为 1 的 1*2 矩阵。 	
b = np.ones((1, 2))
print b
	[[1.　1.]]

# 创建各元素值为 7 的 2*2 矩阵。 	
c = np.full((2, 2), 7,)
print c
	[[7.　7.]
	[7.　7.]]

# 创建 3*3 的单位矩阵。 	
d = np.eye(3)
print d
	[[1.　0.　0.]
	[0.　1.　0.]
	[0.　0.　1.]]

# 使用随机数创建 3*3 的矩阵。 	
e = np.random.random((3, 3))
print e
	[[1.31744138e-01　5.75103263e-02　8.14373864e-01]
	[6.38513903e-01　5.77462977e-01　5.26181855e-04]
	[8.57136438e-02　2.80388443e-01　6.97968482e-01]]

● 数组索引

和 Python 列表类似，NumPy 数组也可以使用切片语法，因为数组可以是多维的，所以必须为每个维度指定好切片，如下列代码所示。

输入：	输出：

import numpy as np
# 创建秩为 2，形状为(3,4)的数组。 	
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print a
	[[1　2　3　4]
	[5　6　7　8]
	[9 10 11 12]]

# 取出第 1 行，2 列开始的，形状为(2,2)的子数组。 #　[[2 3] #　[6 7]] b = a[: 2, 1 : 3] print b	[[2 3] [6 7]]

切取的子数组实际上是原数组的一份浅备份，因此修改子数组，原始数组也将受到修改，如下列代码所示。

输入：	输出：
print '原始 a:', a[0, 1]	原始 a: 2
b[0, 0] = 77 #　b[0, 0]和 a[0, 1]共享同一内存。 print '修改 b 后的 a：', a[0, 1]	修改 b 后的 a：77

也可以混合整数索引以及切片索引访问数组，但是这会生成一个**秩少于原始数组**的子数组。注意这和 MATLAB 处理的数组切片有些不同，NumPy 有两种数组切片方式，一是混合整数索引和切片，生成低秩子数组；二是仅使用切片，生成与原始数组同秩的子数组。

输入：	输出：

# 创建形状（3,4）秩为 2 的 numpy 数组。 	
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print a
	[[1　2　3　4]
	[5　6　7　8]
	[9 10 11 12]]

row_r1 = a[1, :] # 秩为 1，数组 a 的第二行子数组。
row_r2 = a[1 : 2, :] # 秩为 2，数组 a 的第二行子数组。
row_r3 = a[[1], :] # 秩为 2，数组 a 的第二行子数组。
print '秩为 1：',row_r1, row_r1.shape
	秩为 1：　[5 6 7 8] (4L,)

print '秩为 2：',row_r2, row_r2.shape	秩为 2：　[[5 6 7 8]] (1L, 4L)
print '秩为 2：',row_r3, row_r3.shape	秩为 2：　[[5 6 7 8]] (1L, 4L)

# 作用在列上同样适用: 	
col_r1 = a[:, 1] # 秩为 1，数组 a 的第二列子数组。
col_r2 = a[:, 1 : 2] # 秩为 2，数组 a 的第二列子数组。
print '秩为 1：',col_r1, col_r1.shape
	秩为 1：　[2　6 10] (3L,)

print '秩为 2：'	秩为 2：
print col_r2, col_r2.shape	[[2] [6] [10]] (3L, 1L)

- 整型数组索引

当我们使用切片索引数组时，得到的总是原数组的子数组，而整型数组索引允许我们利用其他数组中的数据构建一个新的数组。如下列代码所示。

输入：	输出：
a = np.array([[1, 2], [3, 4], [5, 6]]) # 整型数组索引示例。 # 返回的数组形状为(3,)。 print a[[0, 1, 2], [0, 1, 0]]	[1 4 5]
# 上述整型数组索引与下列索引相等： print np.array([a[0, 0], a[1, 1], a[2, 0]])	[1 4 5]

# 当使用整型数组索引时，可以重复索引同一个数组元素： print a[[0, 0], [1, 1]]	[2 2]
# 上述整型数组索引等于下列索引： print np.array([a[0, 1], a[0, 1]])	[2 2]

整型数组索引的一个小技巧是从矩阵的每一行中选择或改变元素，如下列代码所示。

输入：	输出：

# 创建新数组用于选择元素。 	
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
print a
	[[1 2 3]
	[4 5 6]
	[7 8 9]
	[10 11 12]]

# 创建数组索引。 	
b = np.array([0, 2, 0, 1])
# 使用数组 b 中的索引选择矩阵 a 每一行中的特定元素。 	
print a[np.arange(4), b] # 打印 "[1 6 7 11]"。
	[1 6 7 11]

# 使用数组 b 中的索引改变矩阵 a 每一行中的特定元素。 a[np.arange(4), b] + = 10 print a	[[11 2 3] [4 5 16] [17 8 9] [10 21 12]]

- 布尔型数组索引

布尔型索引可以任意挑选数组中的元素，这种类型索引频繁地用于条件语句下的元素选取。如下列代码所示。

输入：	输出：
import numpy as np a = np.array([[1, 2], [3, 4], [5, 1]]) print a	[[1 2] [3 4] [5 1]]
bool_idx = (a > 2) # 寻找大于 2 的数组元素，返回相同形状的布尔型数组， # 其每个元素为 a > 2 的布尔值。 print bool_idx	[[False False] [True True] [True False]]
# 我们可以使用布尔数组索引去构造一个秩为 1 的数组， # 其元素与布尔数组的真值相对应。 print a[bool_idx]	[3 4 5]
# 我们也可以将上述内容简洁地用一行语句表达： print a[a > 2]	[3 4 5]

- 数据类型（Data Types）

NumPy 提供了大量的数据类型去构造数组，NumPy 会尝试猜测你创建的数组的数据类

型，但构造函数的数组通常也可选择显式指明其数据类型。如下列代码所示。

输入：	输出：
x = np.array([1, 2]) # 让 numpy 自己选择数据类型。 y = np.array([1.0, 2.0]) # 让 numpy 自己选择数据类型。 z = np.array([1, 2], dtype = np.int64) # 显式的规定数据类型。 print x.dtype, y.dtype, z.dtype	int32 float64 int64

更多有关数据类型详细的说明及其用法，有兴趣的读者可以访问如下网址来进行学习：http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html。

1.3.4.2　数组运算

数组中基本的数学运算操作是按数组元素进行的，并且重载操作以及函数都可以使用，如下列代码所示。

输入：	输出：
x = np.array([[1, 2],[3, 4]], dtype = np.float64) y = np.array([[5, 6],[7,8]], dtype = np.float64) # 按元素求和；以下两种方式都可以使用； print x + y	[[6. 8.] [10. 12.]]
print np.add(x, y)	[[6. 8.] [10. 12.]]
# 按元素相减； print x - y	[[-4. -4.] [-4. -4.]]
print np.subtract(x, y)	[[-4. -4.] [-4. -4.]]
# 按元素乘； print x * y	[[5. 12.] [21. 32.]]
print np.multiply(x, y)	[[5. 12.] [21. 32.]]
# 按元素除； print x / y	[[0.2 0.33333333] [0.42857143 0.5]]
print np.divide(x, y)	[[0.2 0.33333333] [0.42857143 0.5]]
# 按元素取平方根； print np.sqrt(x)	[[1. 1.41421356] [1.73205081 2.]]

注意：和 MATLAB 不同，*在 NumPy 中是按元素乘，而在 MATLAB 中是矩阵乘。在 NumPy 中我们使用 dot 函数计算向量内积（点积）、矩阵乘矩阵及矩阵乘向量等操作。dot 可以当作函数在 NumPy 中使用，也可作为数组对象的实例方法，如下列代码所示。

输入：	输出：
x = np.array([[1, 2],[3, 4]]) y = np.array([[5, 6],[7, 8]]) v = np.array([9, 10]) w = np.array([11, 12]) # 向量内积；都将生成 219。 print v.dot(w)	219
print np.dot(v, w)	219
# 矩阵 / 向量乘积；都将生成秩为 1 的数组[29 67]。 print x.dot(v)	[29 67]
print np.dot(x, v)	[29 67]
# 矩阵 / 矩阵乘积；都将生成秩为 2 的数组。 # [[19 22] # [43 50]] print x.dot(y)	[[19 22] [43 50]]
print np.dot(x, y)	[[19 22] [43 50]]

NumPy 还提供了许多有用的数组计算函数，其中最常用的是 sum 函数。

输入：	输出：
x = np.array([[1, 2],[3, 4]]) print np.sum(x) # 计算所有元素的累加和；打印"10"。	10
print np.sum(x, axis = 0) # 计算每一列的累加和；打印"[4 6]"。	[4 6]
print np.sum(x, axis = 1) # 计算每一行的累加和；打印"[3 7]"。	[3 7]

更多有关 NumPy 的数学函数的使用，感兴趣的读者可以参考如下网址来进行学习：http://docs.scipy.org/doc/numpy/reference/routines.math.html。

除了使用数组进行数学计算，我们还频繁地使用 reshape 或者其他方法操纵数组数据。例如要转置一个矩阵，简单地使用数组对象的 T 属性即可，如下列代码所示。

输入：	输出：
print x	[[1 2] [3 4]]
print x.T	[[1 3] [2 4]]
v = np.array([[1, 2, 3]]) print v	[[1 2 3]]
print v.T	[[1] [2] [3]]

1.3.4.3 广播（Broadcasting）

广播提供了强大的机制，允许 NumPy 在不同形状的数组中执行数学操作。我们经常会遇到小数组和大数组相乘的情况，比如图片数据矩阵与权重矩阵。使用广播机制可以提高代码质量及运算效率。例如要在矩阵的每一行中都加上一个常数向量，可以按如下代码进行操作。

输入：	输出：

# 矩阵 x 的每一行加上向量 v，将结果存储在矩阵 y 中。 	
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x) # 创建一个和 x 形状相同的空矩阵。
# 使用显式循环完成上述操作。 	
for i in range(4):
    y[i, :] = x[i, :] + v
print y
	[[2 2 4]
	[5 5 7]
	[8 8 10]
	[11 11 13]]

这样操作是可行的，但当矩阵 ***X*** 特别大时，在 Python 中计算显式循环就将变得非常缓慢。其实将向量 ***v*** 加到矩阵 ***X*** 的每一行就相当于将向量 ***v*** 拷贝多次垂直堆叠成矩阵 ***VV***，然后对矩阵 ***X*** 与矩阵 ***VV*** 进行按元素求和操作，也可以实现同样的结果，如下列代码所示。

输入：	输出：
vv = np.tile(v, (4, 1)) # 拷贝 4 次向量 v，然后将其堆叠起来。 print vv # 打印 "[[1 0 1] # [1 0 1] # [1 0 1] # [1 0 1]]"	[[1 0 1] [1 0 1] [1 0 1] [1 0 1]]
y = x + vv # 矩阵 x 和矩阵 vv 按元素相加。 print y	[[2 2 4] [5 5 7] [8 8 10] [11 11 13]]

NumPy 广播机制允许我们在不创建多次向量 ***v*** 备份的情况下执行该计算，如下列代码所示。

输入：	输出：

import numpy as np
# 矩阵 x 的每一行加上向量 v，将结果存储在矩阵 y 中。 	
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v # 使用广播将 v 加到矩阵的每一行上。
print y
	[[2 2 4]
	[5 5 7]
	[8 8 10]
	[11 11 13]]

由于广播机制的原因，即使 ***X*** 的形状为(4,3)，***v*** 的形状为(3,)，表达式 $y = x + v$ 依然可以

执行；这就好像将 ***v*** 拷贝重塑为(4,3)的矩阵，然后进行按元素相加。对两个数组使用广播机制要遵守下列规则。

1．如果数组的秩不同，将秩较小的数组进行扩展，直到两个数组的尺寸长度都一样。

2．如果两个数组在某个维度上的长度是相同的，或者其中一个数组在该维度上的长度为1，那么我们就说这两个数组在该维度上是相容的。

3．如果两个数组在所有维度上都是相容的，它们就能使用广播。

4．广播之后，两个数组的尺寸将和较大的数组尺寸一样。

5．在任何一个维度上，如果一个数组的长度为 1，另一个数组长度大于 1，那么在该维度上，就好像是对第一个数组进行了复制。

如果感觉没有解释清楚，更多关于广播的详细说明文档可以参考以下网址：

http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html 或者更具体的解释可以访问 http://wiki.scipy.org/EricsBroadcastingDoc 网址。

支持广播机制的函数也被称为**通用函数**（Universal Functions），可以访问 http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs 网址来查看所有的通用函数。以下是广播的一些应用。

输入：	输出：

# 计算向量外积。 	
v = np.array([1, 2, 3]) # v 形状(3,)。
w = np.array([4, 5]) # w 形状(2,)。
# 要计算外积，我们首先要重塑 v 为一列(3, 1)，然后将其与 w(2,)相乘， 	
# 输出一个形状为(3,2)的矩阵，其就是 v 与 w 的外积。 	
print np.reshape(v, (3, 1)) * w
	[[4 5]
	[8 10]
	[12 15]]

# 将向量加到矩阵中的每一行。 	
x = np.array([[1, 2, 3], [4, 5, 6]])
# x 形状(2, 3)，v 形状(3,)，广播之后的形状(2, 3), 	
print x + v
	[[2 4 6]
	[5 7 9]]

# 将向量加到矩阵的每一列。x 形状(2, 3)，w 形状(2,)， # 如果我们将 x 进行转置，其形状就被重塑为(3, 2)， # 然后将其与 w 进行广播，就可以生成形状为(3, 2)的矩阵； # 最后再将结果进行转置，就可以得到形状为(2, 3)的矩阵； # 这个矩阵就是将向量 w 加到矩阵 x 的每一列所得到的结果。 print (x.T + w).T	[[5 6 7] [9 10 11]]
# 另一种更简单的方法是将 w 重塑为形状为(2,1)的行向量； # 然后直接与 x 进行广播就可产生相同的结果， # 注意(2,)表示的是秩为 1 的向量，(2,1)表示的是秩为 2 的矩阵。 print x + np.reshape(w, (2, 1))	[[5 6 7] [9 10 11]]
# 矩阵各元素乘以一个常数， # x 形状(2, 3)，Numpy 将标量作为形状为( )的数组进行处理； print x * 2	[[2 4 6] [8 10 12]]

通过上面内容的介绍，发现广播可以使代码简洁而高效，因此应该尽可能地使用广播操作。以上仅仅是 NumPy 一些重要的用法，但其功能远不只这些。详细的文档请参考网址：http://docs.scipy.org/doc/numpy/reference/。

1.3.5　Matplotlib

Matplotlib 是一个绘图工具库。下面我们简单地介绍下 matplotlib.pyplot 模块，其用法和 MATLAB 相似。

```
import matplotlib.pyplot as plt
# 使用以下 IPython 命令行，可以使得绘图结果嵌入到 notebook 中。
%matplotlib inline
```

- 绘制（Plotting）

Matplotlib 最重要的函数就是绘制函数 plot，使用 plot 函数可以绘制 2D 数据，如下列代码所示，绘制好的图形如图 1-2 所示。

```
# 使用 sin 三角函数计算 x 与 y 的坐标点。
x = np.arange( 0, 3 * np.pi, 0.1 )
y = np.sin( x )
# 使用 plot 函数绘制坐标点。
plt.plot( x, y )
```

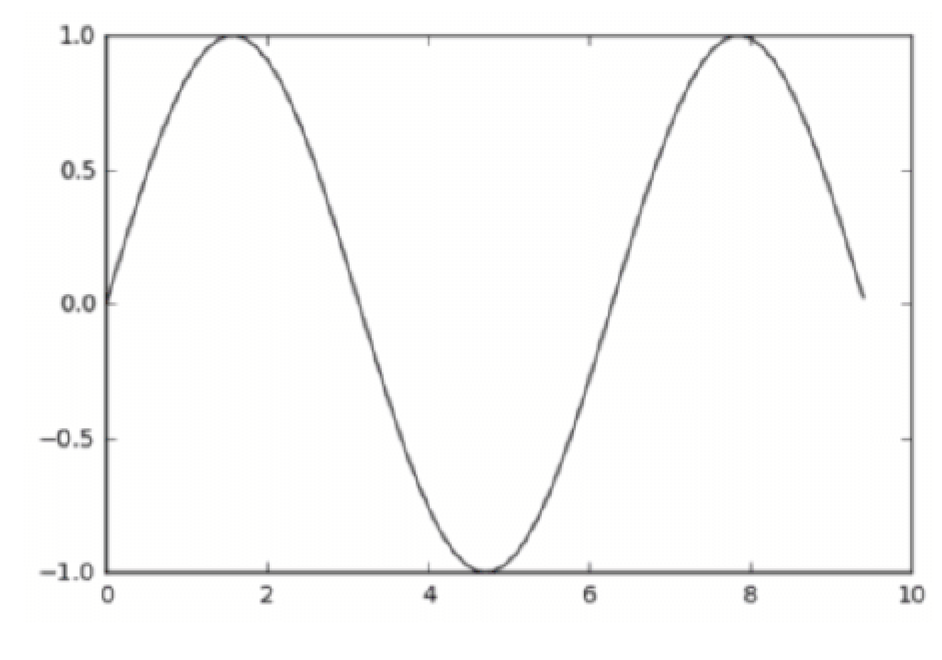


图 1-2　使用 plot 函数绘制图形

添加标题、说明及坐标轴标记到图表中，如下列代码所示，绘制好的图形如图 1-3 所示。

```
x = np.arange( 0, 3 * np.pi, 0.1 )
y_sin = np.sin( x )
y_cos = np.cos( x )
# 使用 plot 函数绘制坐标点。
plt.plot( x, y_sin )
plt.plot( x, y_cos )
plt.xlabel( 'x axis label' )
```

```
plt.ylabel( 'y axis label' )
plt.title( 'Sine and Cosine' )
plt.legend( [ 'Sine', 'Cosine' ] )
```

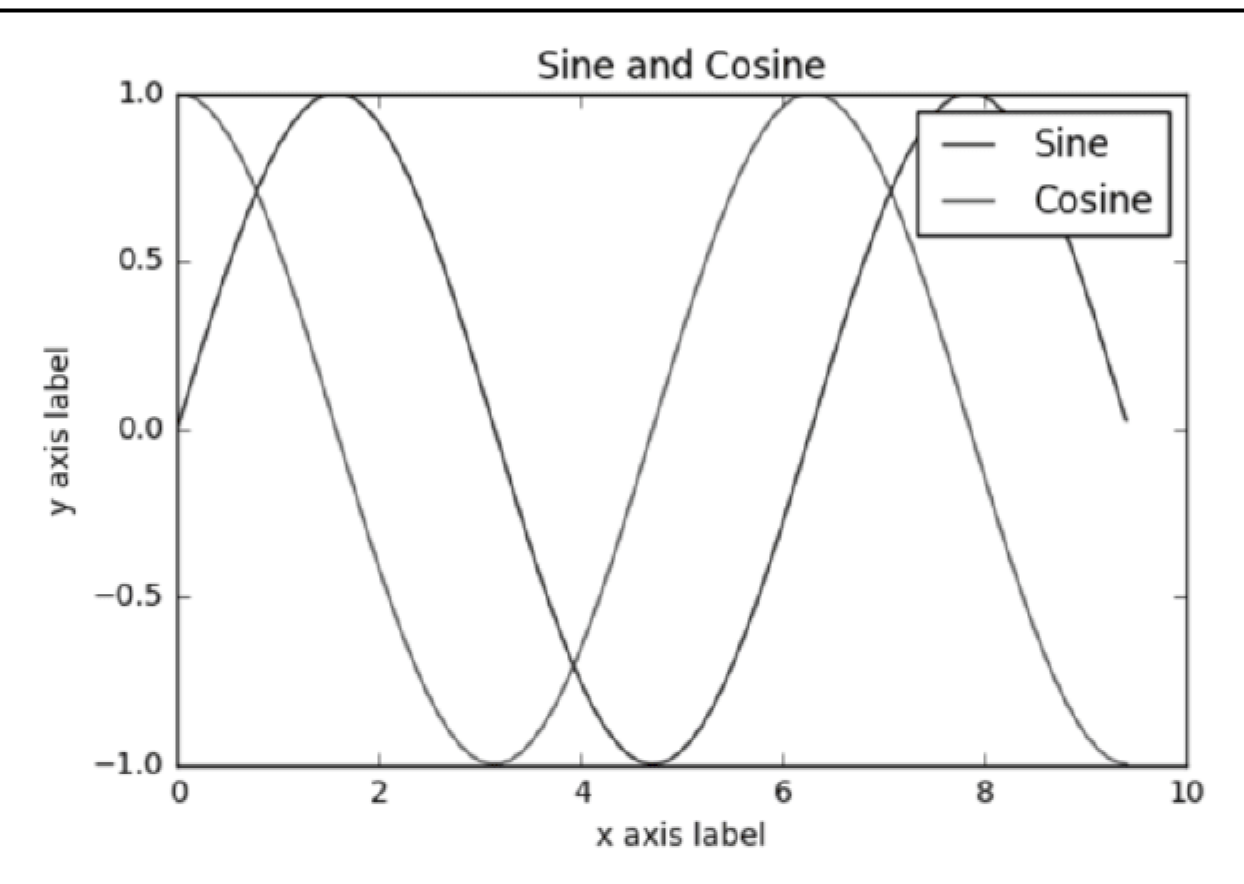


图 1-3　添加标题、说明与坐标轴标记

- 子图（Subplots）

还可以使用 subplot 函数在一幅图中绘制不同的子图，如下列代码所示，绘制的子图如图 1-4 所示。

```
# 使用 sin 以及 cos 函数计算 x 与 y 的坐标点。
x = np.arange( 0, 3 * np.pi, 0.1 )
y_sin = np.sin( x )
y_cos = np.cos( x )
# 设置子图网格，其高为 2，宽为 1。
# 设置使用第一张子图。
plt.subplot( 2, 1, 1 )
# 绘制第一张子图。
plt.plot( x, y_sin )
plt.title( 'Sine' )
# 设置使用第二张子图，并绘制。
plt.subplot( 2, 1, 2 )
plt.plot( x, y_cos )
plt.title( 'Cosine' )
# 显示图表。
plt.show( )
```

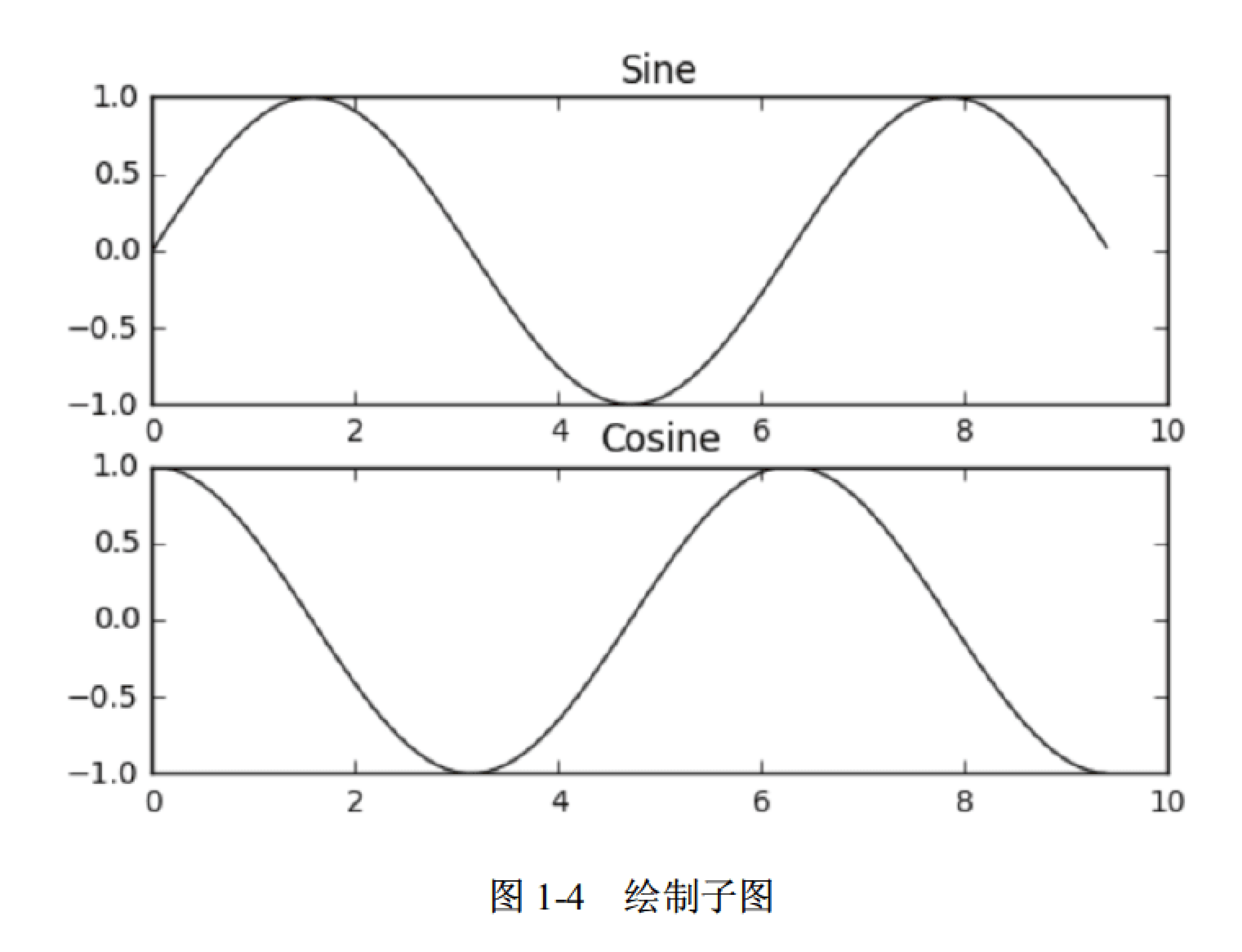


图 1-4　绘制子图

更多关于 Matplotlib 的内容及其使用，感兴趣的读者可以访问以下网址：http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.subplot 来进行学习。

1.4　参考文献

[1] Feigenbaum, E. A., & Feldman, J. (1950). Computing machinery and intelligence. Mind, 6(137), 44-53.

[2] Lecun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.

[3] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning rep-resentation by back-propagating errors. Nature, 323(3), 533-536.

[4] Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. Science, 313(5786), 504-507.

[5] Bengio, Y. (2009). Learning Deep Architectures for AI: Now Publishers.

[6] Fodor, J. A., & Pylyshyn, Z. W. (1988). Connectionism and cognitive architecture: a critical analysis. Cognition, 28(1–2), 3-71.

[7] Jain, A. K., Mao, J., & Mohiuddin, K. M. (1996). Artificial Neural Networks: A Tutorial. Computer, 29(3), 31-44.

[8] Mcculloch, W. S., & Pitts, W. H. (1943). A logical calculus of ideas imminent in nervous activity.

[9] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain: MIT Press.

[10] Widrow, B. (1960). Hoff: Adaptive switching circuits.

[11] Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics, 36(4), 193-202.

[12] Lecun, Y., & Bengio, Y. (1998). Convolutional networks for images, speech, and time

series: MIT Press.

[13] Touretzky, D. S., & Hinton, G. E. (1988). A Distributed Connectionist Production System. Cognitive Science, 12(3), 423-466.

[14] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1988). Learning internal representations by error propagation: MIT Press.

[15] Hofmann, T., Schölkopf, B., & Smola, A. J. (2008). Kernel Methods in Machine Learning. Annals of Statistics, 36(3), 1171-1220.

[16] Heckerman, D. (1992). A tutorial on learning with Bayesian networks: MIT Press.

[17] Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2006). Greedy layer-wise training of deep networks. Paper presented at the International Conference on Neural Information Processing Systems.

[18] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. Paper presented at the International Conference on Neural Information Processing Systems.

第 2 章

机器学习快速入门

深度学习只是机器学习的重要分支之一。为了更好地理解深度学习，我们首先简单介绍一些重要的机器学习原理与思想，而这些内容也将贯穿我们整个深度学习之旅。该章节内容是不全面的，但这些内容全都和深度学习内容高度相关。因此，对于想要了解更全面的机器学习知识的读者，我们强烈推荐周志华老师所著的《机器学习》[1]一书，该书是国内机器学习的殿堂级著作，并且主要针对中国读者，读者会少一些对于翻译的不适应。国外也有许多经典的机器学习著作，如 Tom Mitchell 的 *Machine Learning*[2]，Bishop 的 *Pattern Recognition and Machine Learning*[3]和 Murphy 的 *Machine learning: A Probabilistic Perspective*[4]，这些都是非常经典的机器学习教材。如果你已经非常熟悉机器学习的基础内容，可以轻松地跳过该章节，仅仅完成 2.6 节的 Softmax 编码练习即可。但话又说回来，再看看原本已经非常熟悉的内容，是一件愉快身心的事情，为什么不让自己开心一下呢？

机器学习允许我们去处理那些我们很难通过直接编程实现的任务。“一花一世界，一叶一菩提”，我们以一种更加哲学略带禅意的观点来看，要理解并研究机器学习，其本质上也是对我们自身潜在智能行为的深入理解。而理解这些智能行为，最好的教材就是我们自己。我们就是自己的老师、教材及学生。因此在学习机器学习算法时，充分发挥想象力，想想自己的行为，或许可以给你带来许多豁然开朗的欣喜。

要让机器能够学习，那首先得知道**什么是机器学习**。因此在 2.1 节中，我们将介绍一些机器学习任务，讲解如何去度量机器学习算法性能的好坏。

机器学习的目标是什么呢？那就是降低代价函数。在 2.2 节中，我们将讲解什么是**代价**

函数（损失函数），而本章我们将介绍最具代表性的**均方误差**和**极大似然**两种代价函数。

有了目的，接下来如何去做才是关键。在 2.3 节中，我们将讲解如何去降低代价函数，我们只使用一招，也是大家在高中就非常熟悉的一招，那就是**函数求导**。

生活中是否有很多事情总觉得“做得太少”，总有种“如果当初多学些，当初多考虑些，那结果可能会更好”的错觉；而又有很多事情觉得已经“做得太多”，总会郁闷于“明明万事巨细了，明明每一个细节都考虑了，但结果和自己的预期却相差甚远”。你有这些问题，当然机器也有，有些人因为过于愚钝而变得“愚蠢”，有些人因为过于“精明”而看不开，所谓大学之道在于**中庸**。在 2.4 节中，我们将讲解**过拟合与欠拟合**问题，聊聊机器也会过于“笨拙”或者过于“聪明”，并如何“中庸”地解决这些问题。

你是否痛恨考试，听到考试就会瑟瑟发抖，但为什么要考试呢？生活其实就是一场一场的考试，只不过并不是每次都用试卷和分数去衡量你的好坏，今天你期望着考试的结束，明天或许就被各种业绩考核压得喘不过气来。如果考试是“结束”，那么在“大审判”来临前我们能做些什么呢？在 2.5 节中，我们将讲解如何在“考试”前做些“自我测试”，这些“测试”我们称为**验证集**，我们将调整**超参数**，然后试图使测试成绩更好，从而期望考试成绩也能更好。

道理我都懂，就是不会做。但别怕，在 2.6 节中，我们活动一下手指，再结合整章的知识，一步一步地完成 **Softmax** 多分类图像任务，可以让你信心倍增。

2.1 学习算法

机器学习算法可以简单理解为是一种能够**从数据中学习**的算法。那么什么是学习呢？我们使用 Mitchell（1997）提出的定义：“对于某类**任务** T（Task）和**性能度量** P（Performance Measure），如果一个计算机程序在某项任务 T 中，其性能 P 能够随着**经验** E（Experience）而自我完善，那么我们就称这个计算机程序在从经验中学习。”

如果你对于以上的定义有些不太理解，没关系，那本来也只是让你去大体了解一下而已，其实学习就是一个逐渐优化的过程，我们需要不断地体会与领悟。讲一个少年小飞的故事，小飞和小鱼是男女朋友，关系还算融洽，经常会一起吃饭。每次约好一起去吃饭，小飞都会兴冲冲地跑去等小鱼，但每次都要等很久，小飞也不敢抱怨，但也不想傻傻地站在楼下，两手插在口袋里等待，于是他就开始了“学习”。小飞的任务是“缩小等待的时间”，于是在约好吃饭后就心慌慌地等待了 5 分钟后再出门，结果还是在楼下等了 15 分钟。下一次小飞就犹豫地等待 10 分钟后再出门，还是多等了 5 分钟。再一次，小飞就等了 15 分钟后再出门，结果居然还是等了 10 分钟。后来，小飞果断地等了 25 分钟后再出门，结果就被小鱼胖揍了一顿……从以上的小飞悲惨故事中，你是否对于**任务、性能度量和经验**有了点感悟。如果还没有，没关系，旅程还长，你总会知道的。还有一点需要说明就是小飞的学习是无用的，因为**学习的前提是要有潜在规律**，老实地在楼下等待虽然是万全之策，但是小鱼出门时间是没有规律的。

2.1.1　学习任务

我们相对正式地去定义“任务”一词，但要记住，学习并不是任务本身。学习指的是**获得执行任务的能力**。比如，我们想要机器人行走，那么行走就是任务，我们可以让机器人学习行走，也可以直接编程让机器人行走。

机器学习任务通常被描述为一个“范例”或“数据”，然后研究机器学习系统怎么样去处理这个“范例”或“数据”。一条数据是一个特征集合，这些特征度量了需要机器学习系统处理的事情。我们通常将一条数据表示为一个 n 维实数向量 $\boldsymbol{x}\in\mathbb{R}^n$，而 $\boldsymbol{x}_i$ 是该向量的一个特征。如果读者不太理解向量，可以把向量简单地理解成一维数组即可，而该数组中的每一元素就是一个特征。例如，我们想要完成图像识别任务，那一张图片就是一条数据，图片中的每一个像素就是一个特征。机器学习可以解决许多任务，接下来我们列举一些最常见的机器学习任务。

- **分类（Classification）**：分类任务是最常见的机器学习任务之一，其目的是让程序判断一组数据应该属于哪些类别，例如对象识别任务和语音识别任务等。可以说与“识别”一词相关的任务，很大程度上都属于分类任务。要解决这类问题，学习算法通常要执行一个函数 f，该函数将数据映射到一个 k 类别的空间，形式化表示为 $f:\mathbb{R}^n\to\{1,\cdots,k\}$，也可以使用 $y=f(x)$ 表示该映射关系，其中输入数据表示为向量 $\boldsymbol{x}$，输出 y 为一个整数值，表示分类代号。但有时我们并不输出一个离散的 y 值，而会输出实值概率分布。比如，要判断一封邮件是否是垃圾邮件，这是一个典型的二分类任务，输出“1”表示该邮件为垃圾邮件，输出“0”表示该邮件为正常邮件，但也可以输出“0.8”，表示为该邮件为垃圾邮件的概率为 0.8。
- **回归（Regression）**：回归任务就是要求程序根据输入数据来预测一个数值，比如：股市预测任务和天气预测任务等。为了解决这类问题，学习算法需要将 n 维特征映射到一个实数空间，表示为 f: $\mathbb{R}^n\to$R，除了输出格式的不同，这类任务和分类任务没有本质区别。
- **机器翻译（Machine translation）**：该任务是深度学习领域中最激动人心的任务之一，目前深度学习在图像识别领域取得了重大突破，但很多学者认为视觉虽然非常复杂，但这是大多数动物都具有的基础性能力。可是语言不一样，语言是人这一类比较高级的生物才具有的能力，因此有人断言，深度学习很难在自然语言处理领域取得较大突破。深度学习在自然语言处理领域的突破，造成了深度学习研究者与自然语言处理研究者的激烈讨论，有时甚至恶化到互相排斥的冲突，但在嘈杂之下，也不断地激发人们去深思“智能”的内涵。在机器翻译任务中，输入的数据是由序列信号组成的语言构成，计算机程序需要将该组序列转换成另一种语言序列信号，比如我们将英语转换为中文。这种转换可以是文本转换，也可以是音频的转换。
- **结构化输出（Structured output）**：结构化输出是指我们的输出结果为一个向量，这是一类宽泛的任务，前面提到的机器翻译就属于其中。图 2-1 为 Google 做的一项融合图像识别与机器翻译的图像说明应用，该应用在网络的前几层使用卷积神经网络提取图像特征，然后再将这些特征输入进循环神经网络，将图像特征转换为文字信

号输出。其输入为一张图片，而输出为图片的文字描述。

- **异常检测（Anomaly detection）**：生活中有很多事情并不常见，但我们却很担心这种小概率事件的发生，异常检测就是用程序检测这些离群数据。比如：金融领域中信用卡欺骗越来越常见，由此造成的损失一向难以估计。这些欺诈行为非常严重地扰乱了正常的市场经济秩序，同时也损害了公司以及诚信消费者的切身利益。这类任务的困难在于异常数据相对正常数据而言太少，甚至有时还需要解决未曾发生过的状况。
- **降噪（Denoising）**：在机器学习中，我们总是假定已知数据和未知数据是相似的，我们在已知数据中找寻各种规律、各种模式，然后将已知数据中学习到的模式应用到未知数据中。但在某种程度上，其实是在“自欺欺人”，因为已知数据和未知数据之间是有噪声的，这些噪声造成了实验结果和实际结果的误差。机器学习的一个重要研究方向就是让机器拥有强大的抗噪声能力去对抗这些误差。为了获得这种健壮性，我们在数据中加入一些噪声学习，从“污浊”的数据中还原“干净”的数据，从而期望学习模型能够学习到更强的抗噪能力。

以上介绍的仅仅是机器学习任务中的沧海一粟，我们只列举了一些常见的机器学习任务。其实说到底，机器学习就是**试图替代或辅助人的智能行为**，或许人的智能所触及之处，都将是机器学习努力的方向。



一个女人在公园里扔**飞盘**



一只**狗**爬在硬木地板上



一个**小女孩**抱着泰迪熊坐在床上



一群人坐在船里

图 2-1　图片说明应用示例

2.1.2　性能度量

为了评估机器学习算法在某项任务中的好坏，我们就必须设计方法去量化其性能。比如

在分类任务中，我们经常衡量模型的**精度**（accuracy）。精度其实就是正确分类数据与全部分类数据的比值。与之相对应，我们去测量错误分类数据在全部数据中的比例，就将其称之为**错误率**（error rate）。我们也经常将错误率称为 0-1 损失期望。在 0-1 损失中，我们将分类错误的数据记为 1，而分类正确的数据就记为 0，累加错误分类数据之后，再除以全部的分类数据，就得到了 0-1 损失期望值。

机器学习算法是要在实际环境中运行的，也就是说，机器学习所面临的数据是**未知的**。就像我们人一样，学得再多，也只是纸上谈兵，实践才能出真知。但未来总是复杂多变的，庄子曾说过"吾生也有涯，而知也无涯，以有涯随无涯，殆已。"那我们又怎样固执地从已知中，头破血流地去追寻未知呢？因此我们需要"假造"一些未知数据，我们称之为**测试数据集**（test set of data），我们将训练好的机器学习算法拿到这些测试数据集上进行性能测量，然后我们就"自欺欺人"地宣称我们设计的算法如何的好。测试数据其实也是已知数据的一部分，只是这部分数据我们会密封起来，只在最终的性能测量时才使用。如果拿课程学习作为类比，那平时自己做的练习题或家庭作业就是我们的**训练数据**（training set of data），而老师组织的月考、期中考，我们也称为验证数据（在本章 2.5 节中详细描述），最后参加的期末考试我们就称之为测试数据。这三部分数据全是已有的数据，其中训练数据只用来训练，学生（学习算法）的目标就是在这部分数据上不断提高性能，而老师（算法调整人员）会使用验证数据监控学生的学习情况，然后去调整学生的学习方式。当老师认为该学生已经无法再提高时，就使用测试数据模拟未知数据对学生进行最后的性能测试。

- 查全率与查准率

错误率和精度是最常用的度量方式，但在特定的任务中，我们还需要一些额外的度量方式。比如进行信息检索时，我们经常需要关心"检索的信息中有多少是用户感兴趣的"以及"用户感兴趣的信息有多少被检索出来了"这两类问题，而此时，我们引入**查准率**（precision）与**查全率**（recall）。

为了更好地介绍查准率与查全率，以二分类问题为例，我们需要将分类器预测结果分为以下 4 种情况。

（1）真正例（True Positive，TP），分类器预测 1，真实类标为"1"的分类数据。
（2）假正例（False Positive，FP），分类器预测 1，真实类标为"0"的分类数据。
（3）真反例（True Negative，TN），分类器预测 0，真实类标为"0"的分类数据。
（4）假反例（False Negative，FN），分类器预测 0，真实类标为"1"的分类数据。
而 TP+FP+TN+FN=数据总量，如表 2-1 所示为**混淆矩阵**（Confusion Matrix）。

表 2-1　混淆矩阵

真实情况	预测结果	
	正例	反例
正例	TP（真正例）	FN（假反例）
反例	FP（假正例）	TN（真反例）

如式（2.1）所示，查准率 P 就是分类器预测结果为"1"中预测正确的比率。

$$P = \frac{TP}{TP + FP} \tag{2.1}$$

查全率 R 就是全部真实类标为“1”的数据中分类器预测正确的比率，如式（2.2）所示。

$$R = \frac{TP}{TP + FN} \tag{2.2}$$

查准率与查全率是一对“鱼”与“熊掌”，一般来说，查准率高时，查全率往往偏低；而查全率高时，查准率往往偏低。例如，如果想要将垃圾邮件都选取出来，可以将所有邮件都标记为垃圾邮件，那查全率就可以接近于 1 了，但这样查准率就会比较低；如果希望分类垃圾邮件的查准率足够高，那么让分类器尽可能挑选最有把握的垃圾邮件，但这样往往会有大量的垃圾邮件成为漏网之鱼，此时查全率就会比较低。

需要注意的是，性能度量也依赖于所需完成任务的应用场景。任务场景不同，度量手段也不相同，如果性能度量选取不当，对于该任务可能是毁灭级的。比如我们让机器完成指纹识别任务，但应用在超市中和应用在安全机构部门的指纹识别系统，其性能度量方式完全不同。在某个超市会员系统中，需要识别用户的指纹以确定用户是否为会员，然后对部分商品进行打折或抽奖服务，此时我们并不关心指纹识别的精确度，假设一个会员用户总是被识别错误，那么该用户就会非常恼火，超市很有可能就会失去一个老顾客。相反，将一个非会员顾客识别错误，其损失就没那么高，或许还有可能获得一个新会员，因此该任务会更关心**查全率**，也就是期望所有会员都要被识别出来。但如果将指纹识别应用在某安全部门，用于识别员工的身份，此时我们就更关心识别员工的正确性，因此机器只会在最有“把握”的情况下才允许员工通过，此时我们更关心**查准率**，我们或许情愿对员工识别错误 100 次，也不愿意放过一个非员工，或许此时机器可能“非常不好用”，但安全性得到了保障。

2.1.3 学习经验

根据数据（经验）的类型，我们可以将机器学习算法粗略地分为**监督学习**（Supervised Learning）和**非监督学习**（Unsupervised Learning）。

监督学习算法：试图将已知数据与该数据所对应的标记或类标（label）进行关联。比如幼儿园老师教小朋友识别树，老师拿一堆图片给小朋友看，并告诉小朋友这些图片里都是树。小朋友就拿着这些图片观察（学习），之后老师就再拿些新的图片给小朋友识别，小朋友根据自己已经学习到的知识去识别新的图片，老师会告诉小朋友哪些图片识别对了，哪些图片识别错了。然后根据老师的纠正，小朋友就再次调整自己，不断地学习提高自己识别树的能力。

在我们所拥有的资料中，既有**数据** x（比如一张张的图片），也有数据所对应的**标记** y（比如老师的指导），我们从（x，y）中学习的过程就叫监督学习。例如，进行数字图像识别任务时，我们将数字图片作为数据输入到机器学习算法中，然后算法会预测出一个数字分类 y'，之后我们依据图片所对应的真实标记 y，比较 y 与 y' 的差异再来调整学习算法。

非监督学习算法：生活中的数据绝大多数是没有标记的，并且也缺少金钱与人力去标记数据。非监督学习就是在没有指导（标记）的前提下，学习数据集内部的有用结构。最常见的非监督学习算法是**聚类**（Clustering），该类算法通常按照中心点或者分层的方式对输入数

据进行归并，试图找到数据的内在结构，以便按照最大的共同点将数据进行归类。

从概率的角度来看，非监督学习试图通过随机变量 x 去学习概率分布 $P(x)$或某些分布性质；监督学习试图使用随机变量 x 以及关联 y，学习条件概率 $P(y|x)$，试图通过 x 去预测 y。但监督学习与非监督学习并不是一种形式化的定义，其界限有时也很模糊。

在一些情况下，我们既有标记数据，也有大量的未标记数据，我们将这两类数据都利用起来就称为**半监督学习**（Semi-supervised Learning）。

有时数据需要从环境中获取，而数据对应的标记也要从环境中获取，我们并不提供标记数据，我们只提供某种评价机制（奖励或惩罚），这样的学习方式称之为**强化学习**（Reinforcement Learning）。就像我们教小狗上厕所，但小狗可听不懂你的话，就需要不停地奖励或惩罚其行为，小狗才可能学会上厕所。

为了统一描述，以后我们会将数据集称之为数据矩阵，也可以简单地想象为二维数组，矩阵的**列为数据的不同特征**，矩阵的**行为不同数据**。例如，我们有 88 张大小为 32×32 的灰度图片，那么数据矩阵就为 $\boldsymbol{X} \in \mathbf{R}^{88\times1024}$，$X_{4,20}$就表示为第 4 张图片的第 20 维像素。

2.2　代价函数

机器学习中绝大多数任务都是优化任务，也就是去寻找最优解。对应到函数中，就是寻找函数极值。但这其中就存在一个很困难的问题，局部最优解。传统的机器学习大多是针对凸优化问题，也就是函数可以找到最大值或最小值，但很不幸，由于深度学习构造的函数非常复杂（模型容量大），因此深度学习会存在很多局部最优解（数据高维时鞍点更普遍），这也是深度学习所面临的最困难的挑战之一，为此我们也将在第 5 章中专门地介绍深度学习的优化技术。

绝大多数机器学习算法学习的过程，其实就是在调整数据特征的重要性，我们将这种刻画重要性的量称之为**参数或权重**，参数控制着机器学习系统的行为，而我们要做的其实就是找到一组最优的参数。我们可以把机器学习算法简单地想成一个函数 $f_\omega(x)$, ω表示函数的参数，x 表示我们输入的数据，$f_\omega(x)$可以是简单的线性回归算法，如式（2.3）所示。

$$f_\omega(x) = \omega_1 x_1 + \omega_2 x_2 + \cdots + \omega_n x_n \tag{2.3}$$

也可以是层层嵌套的深度学习算法，如式（2.4）所示。

$$f_w(x) = \alpha f(\omega_1 f(wx\cdots) + \omega_2 f(wx\cdots)\cdots) \tag{2.4}$$

暂不考虑其内部结构，我们的目的就是尽可能地让我们设计的函数尽可能的“好”。而所谓的“好”就是机器学习算法的预测值与实际值之间的误差尽可能的低，我们也将衡量这种误差的函数称之为**代价函数**（Cost Function）或者**损失函数**（Loss Function）。

2.2.1　均方误差函数

假设需要完成一个房屋价格预测任务，我们需要输入房屋面积、户型、位置等特征，输出为房屋价格预测。我们用集合 $X = \{x^{(1)}, x^{(2)}, \cdots x^{(m)}\}$ 表示已有的 m 条房屋信息（数据），

用集合 $Y=\{y^{(1)},y^{(2)},\cdots y^{(m)}\}$ 表示对应的实际售出价格。并且我们也设计好了机器学习算法 $f(x;w)$，而你需要做的是，思考一下如何设计我们的代价函数。你能想到最简单的代价函数会是什么呢？比如式（2.5）这样一个简单的代价函数。

$$J(w)=\sum_{i=1}^{m}|y^{(i)}-f(x^{(i)};w)| \tag{2.5}$$

你能看出式（2.5）表示什么意思吗？其实很简单，就是把每一个预测的房屋价格与真实的房屋价格相减，因为我们只关心两者的误差于是就加了一个绝对值，然后将所得的误差累加起来就得到上面的公式了。

假如你去预测房屋的价格，你又怎样评价自己误差呢？你会不会这样想“有些预测非常准确，有些预测比较差，但平均下来还是挺不错的。”自然而然，我们去估算一堆事物的好坏，也总是使用平均值，用统计学的术语就叫期望值，我们再稍稍修改式（2.5）中的代价函数就得到了式（2.6）。

$$J(w)=\frac{1}{m}\sum_{i=1}^{m}|y^{(i)}-f(x^{(i)};w)| \tag{2.6}$$

式（2.6）已经足够优秀了，但有一个缺点，那就是有绝对值，因为绝对值不连续也不能处处可导，那么怎样既不破坏我们近乎完美的设计，又可以去掉绝对值呢？我猜你 5 秒钟内就会想到，可以对其求平方，如式（2.7）所示。

$$J(w)=\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}-f(x^{(i)};w))^2 \tag{2.7}$$

就是如此简单，我们轻松地设计出了第一个代价函数，但请你不要误会，式（2.7）代价函数可是大名鼎鼎的**均方误差**（Mean Squared Error）[5]。

看完以上式的推导，不知道你有何感想，其实这些式只是我们各种想法的简化描述，看到式（2.7）会不会有一种豁然开朗的感觉，我们需要用一堆文字描述的思想，只需一行表达式就可以完成。开始时希望你不要急躁，如果对这一过程有些不懂，再折回去慢慢体会这一过程也是个不错的选择。请记住这些式来帮助你记忆并理解上文中的内容，不是你的负担。这些式并不重要，重要的是其所表示的含义。

2.2.2 极大似然估计

“阳光正热，才过午后，学徒小飞沏了一杯下午茶，准备躺在竹摇椅上享受一番。不料杨师傅一记闷拳敲在了头上，小飞顿时头晕眼花，抱头嚎叫。‘劣徒，你还有没有梦想？快和为师上山打猎。’两人走到半山腰，发现了一只小兔子，于是两人同时举起了枪，小白兔就倒在了血泊中。”那么问题来了，是谁残忍地杀害了小白兔？

如果大侠没开脑洞，正常情况下应该会选择杨师傅吧，也许你觉得你的选择很正常没什么奇怪的，但其实你已经用到了**极大似然**的思想。那“似然”又具体指什么呢？**似然**（likelihood）是一种较为文言文的翻译，我们脱去其外衣可能看得比较清楚。“似然”用现代的中文来说就是“可能性”，所以极大似然，通俗点就是最大的可能性。我们选择杨师傅的原因是因为杨师傅帅吗？当然不是，那是因为杨师傅是“老炮”，他的枪法准，射中的可能性更大。

回到生活中，我们几乎处处都在使用“极大似然”，很多时候我们没机会，也没能力去探究事物的真相，我们最简单也是最高效的方法就是去估计。当你去找工作时，别人很有可能会问你，你来自哪所学校？你的文凭如何？你得过什么奖？记住，任何指标都不能确认你是一个多么优秀的人，指标只能确定你是优秀人才的可能性。

在继续讨论“极大似然”之前，我们要补充机器学习中非常重要的一个假设，那就是**数据独立同分布**（independent and identically distributed，i.i.d.）条件。独立同分布假设数据与数据是独立的，就比如信用卡发放任务中，每一张用户填写的表格是相互独立的，但该任务的所有数据又都来自于同一个概率分布函数，这就使得我们模拟出已知数据的概率分布模型，同样也可以运用在未来的任务中。有了 i.i.d.条件，就可以构造极大似然估计。

我们依然给出一个概率函数 $P(y|x;w)$，为了方便，概率函数只进行了二分类，也就是输出“0”或“1”。那么如何去判断 $P(y|x;w)$的好坏呢？很简单，那就是该概率函数在**已有的数据上发生的概率最高**，考虑到数据是相互独立的，因此可以对每一条数据对应的概率函数进行求积，就得到了式（2.8）。

$$L(w)=\prod_{i=1}^{m}P(y^{(i)}\mid x^{(i)};w) \tag{2.8}$$

这个函数就称为似然函数，而我们求出该函数取得最大值时所对应的参数，也就是**极大似然估计**（Maximum Likelihood Estimate）[6]。

但问题在于，求连积是很困难的，不但耗费计算资源，令人头疼的是还非常容易造成内存下溢出。那我们能不能将乘法改成加法呢？可以使用对数函数将乘法转换为加法。如式（2.9）就是对数运算的基本公式之一。

$$\log_a(MN)=\log_a M+\log_a N \tag{2.9}$$

根据式（2.8）和（2.9），我们对似然函数取自然对数就得到了式（2.10）。

$$\ln L(w)=\sum_{i=1}^{m}\ln P(y^{(i)}\mid x^{(i)};w) \tag{2.10}$$

在习惯上，我们比较喜欢求最小值、平均值，因此式（2.10）就变成了式（2.11）。

$$\ln L(w)=\frac{-1}{m}\sum_{i=1}^{m}\ln P(y^{(i)}\mid x^{(i)};w) \tag{2.11}$$

假设机器学习算法 $f(x)$就是概率函数 $P(y|x)$，需要注意的是如果真实的标记 y 是“0”，那么预测的 $f(x)$也应该接近于“0”，但公式又不允许“0”出现，因此在真实值为“0”时，预测值用 $1-f(x)$代替。最后代价函数就如式（2.12）所示。

$$J(w)=\ln L(w)=\frac{-1}{m}\sum_{i=1}^{m}(y^{(i)}\ln f(x^{(i)};w)+(1-y^{(i)})\ln(1-f(x^{(i)};w))) \tag{式 2.12}$$

也许一眼看上去式（2.12）会比较复杂，但其含义也很简单，当真实值为“1”时，我们就使用 $f(x)$，将右边式子的第二项去掉；当真实值为“0”时，我们就使用 $1-f(x)$，将右边式子的第一项去掉。式（2.12）也被称为**交叉熵**（Cross-entropy）[7]。

2.3　梯度下降法

上面的内容中，我们已经构造了两种代价函数，并且这两种函数都是凸函数，也就是都存在最小值。只要求出代价函数取最小值时的参数，我们的任务也就完成了。那函数的极值又怎么求呢？非常简单，当函数切线的斜率为 0，也就是代价函数的导数为 0 时，也就是函数取得极值时（若自变量多于一个，导数也称为梯度）。但这并不好求，有时因为计算量太大，有时甚至根本找不到直接求解函数极值的公式，因此我们选择愚笨一些的方法“走一步，算一步”。

话说徒弟小飞与杨师傅已经抓到了小白兔，休息一会儿，就准备下山了，然天气大变，周围突然起了很浓的雾。此时小飞大叫，“师傅不好，有妖气！”哐当一下，小飞又被师傅一记长拳，“劣徒，休要胡闹，天气突变，恐有大雨，还未收衣，还不随为师下山。”于是师徒二人就急匆匆地下山了。但雾太浓密，早已分不清去路。师徒俩只能环顾四周，摸索一条下山最陡的路，然后走一段距离，再环顾四周，再往最陡的方向前进，到了傍晚，师徒俩终于回到了家中。如图 2-2 所示，我们根据当前数据，求出“下山”最快的方向（梯度），然后往梯度方向走一段距离，再重复相同步骤的迭代方法，就称为**梯度下降法**（Gradient Descent）[8]。

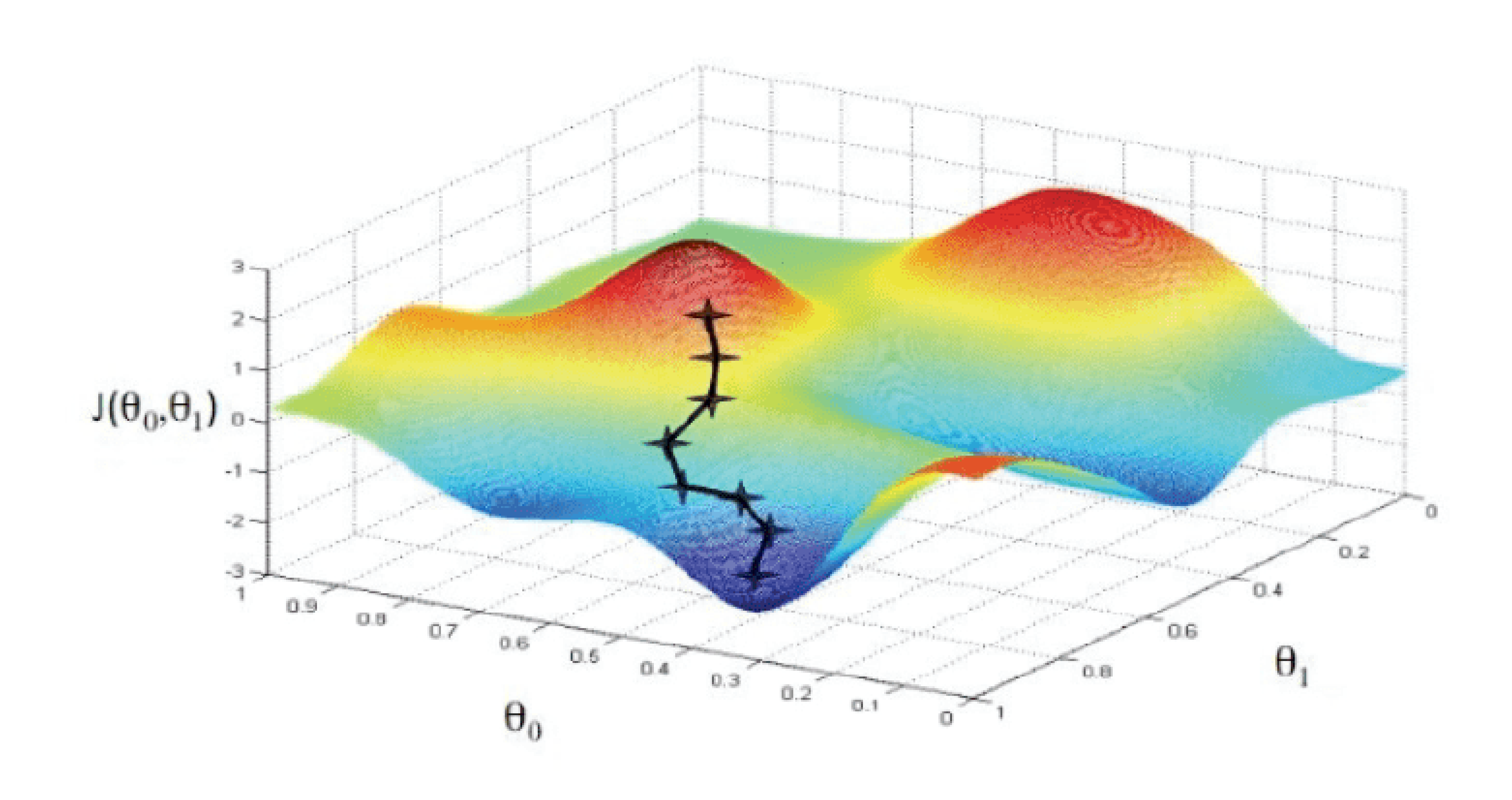


图 2-2　梯度下降示意图

梯度下降法，主要考虑两个问题：一是方向（梯度），二是步长（学习率 α）。方向决定是否走在正确的道路上，而步长决定了要走多久才能到达目的地（错误率最低处）。对于第一个问题，主要集中在如何才能精准地找到最小值的方向。但很遗憾，我们并没上帝视角，因此我们只能找到当前的最佳梯度，但当前的梯度不一定是回家的路，我们却只能前行；对于第二个问题，主要集中在如何确定合理的步长，如果步子太小，则需要很长的时间才能到达目的地，如果步子过大，可能导致在目的地周围来回震荡。

- 方向（梯度）

我们首先来看梯度，求梯度其实就是求多元函数相应变量的偏导数，若参数为一个，也就是数据为一维数据，那么其实就是在求复合函数的导数，如果代价函数为均方误差函数（为

了计算简洁，在代价函数中乘以 0.5），如式（2.13）所示。

$$J(w)=\frac{1}{2m}\sum_{i=1}^{m}(y^{(i)}-f(x^{(i)};w))^2 \tag{2.13}$$

我们的学习器为线性回归，并且数据为二维：$f(x;w)=w_1x_1+w_2x_2$，那么第一个参数 w_1 的梯度就为式（2.14）（注意求导时的负号）。

$$\frac{\partial J(w)}{\partial w_1}=\frac{\partial J(w)}{\partial f(w)}\cdot\frac{\partial f(w)}{\partial w_1}=-\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}-f(x^{(i)};w))\cdot x_1^{(i)} \tag{2.14}$$

同样地，w_2 的梯度就为式（2.15）。

$$\frac{\partial J(w)}{\partial w_2}=\frac{\partial J(w)}{\partial f(w)}\cdot\frac{\partial f(w)}{\partial w_2}=-\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}-f(x^{(i)};w))\cdot x_2^{(i)} \tag{2.15}$$

我们所谓的沿着梯度方向上走一步，其实就是去修改 w：$w_i=w_i-\alpha\frac{\partial J(w)}{\partial w_i}$。

注意：本书是去减梯度，但有些资料中为加梯度，那是在梯度中藏着负号。也可以理解为正确的修改方向是负梯度方向。

虽然上述的推导比较烦琐，如果你能耐心看完，会发现其实非常简单，然后再看看算法 2.1，你会发现我们的关键算法，其实用一行代码就可以完成。因为简单的一行代码，我们讲了一个故事，推导了一系列的公式，虽然有些烦琐，但这些代价都是值得的。

算法 2.1：梯度下降算法用于迭代最优化均方误差代价函数

给定数据集 $X=\{x^{(1)},x^{(2)},\cdots,x^{(m)}\}$，数据集标记 $Y=\{y^{(1)},y^{(2)},\cdots,y^{(m)}\}$，
学习器 $f(x;w)=w_1x_1+w_2x_2+w_3x_3$，学习率（步长）$\alpha$。
For 迭代足够多次
{
$w_i=w_i+\alpha\frac{1}{m}\sum_{j=1}^{m}(y^{(j)}-f(x^{(j)};w))\cdot x_i^{(j)}$
}

- 步长（学习率α）

我们刚刚推导完了梯度，学会了如何找“下山的方向”，现在我们就来聊聊 “下山的速度”。假如小飞是超人，一个奇怪的超人，他的步子可以任意长，但他有一个缺点，那就是比较“耿直”，只会沿着要走的方向直行。山谷那站着他喜欢的人小鱼，他要去见她。他离最低处 4.5m，步长为 1m，当他走到第 5 步时就出现了问题，那就是他多走了 0.5m，接下来他会怎么做呢？根据上面的梯度求法，小飞知道现在的下山方向在他的后面，那他就往后再走 1 米，但接下来他就尴尬了，他就像一个可爱的钟摆一样，不停地在小鱼周围来回震荡，不管小飞走多久，他与小鱼始终相差着 0.5m，此时耿直的小飞眼角都湿润了。你愿意帮帮他吗？请记住，我们从上帝视角知道小飞离小鱼的距离，但旁观者清，当局者迷，小飞是不知

道距离的。

聪明的你可能有很多好的想法，而我就只能想出两个。首先就是提示小飞，“小伙子，欲速则不达，慢一点吧！”，于是小飞就减小了步长，一次走 0.4m，那小飞离小鱼最近的距离也就缩短到 0.1m。而尝到甜头的小飞就把自己的步长缩小到了 0.04m，那他最终离最低点也就更近了。但这里就出现了一个问题，减小步长确实会离成功更近，但时间的代价，也会让小飞心碎。

考虑到刚才的情况，我又有了一个不错的想法。开始时就勇敢地大步往前迈，走到一定步数后就缩小步长。在实际的应用中，步长（学习率 α ）的选择是一个很有技巧性的活，具体情况还要具体分析。总体情况就是 α 过大，代价函数就会在最低值附近较大震荡，而 α 较小代价函数的震荡也会较小，但到达震荡区间的时间也会变长，这也是一个鱼与熊掌的问题。

2.3.1　批量梯度下降法

批量梯度下降（Batch Gradient Descent，BGD）[9]又称最速梯度下降，使用“批量”一词可能会产生歧义，“批量”一词应该是“一批一批处理”的意思，如管道和缓冲器一般，但这里的“批量”其实指的是“全部一起处理”的意思。我们要处理什么呢？我们要处理的其实是数据，前文中说到我们要找“下降**最快**的方向”，因此就进行了求导，但其实我们只是找到了下降的方向，还没找到下降最快的方向。如算法 2.2 所示，才是下降最快的方向，不知道你喜不喜欢玩“大家来找茬”游戏，我们现在就找找算法 2.1 与算法 2.2 的茬吧！

算法 2.2：最速梯度下降算法用于迭代最优化均方误差代价函数

给定数据集 $X=\{x^{(1)},x^{(2)},\cdots,x^{(n)}\}$，数据集标记 $Y=\{y^{(1)},y^{(2)},\cdots,y^{(n)}\}$，
学习器 $f(x;w)=w_1x_1+w_2x_2+w_3x_3$，学习率（步长）$\alpha$。
For 迭代足够多次
{
$w_i=w_i+\alpha\frac{1}{N}\sum_{j=1}^{N}(y^{(j)}-f(x^{(j)};w))\cdot x_i^{(j)}$
}

可以发现，我们把算法 2.1 中的 m 换成了算法 2.2 中的 N，那 N 是什么呢？N 表示的是数据个数，我们计算一次梯度时，需要计算完所有数据中的误差梯度，然后累加之后再取平均值，这就是我们要找的最速下降梯度，也就是我们上文提到的一个信息“环顾四周”。

事物具有两面性，最速梯度下降也具有，优点是准确，我们找到了最正确的方向；缺点就是慢，每计算一次梯度都要遍历所有数据，考虑到如今的大数据时代，这就成了巨大的困难。

2.3.2　随机梯度下降法

随机梯度下降（Stochastic Gradient Descent，SGD）[10]是批量梯度下降的一个极端版本，如算法 2.3 所示，是随机梯度下降的算法描述。

算法 2.3：随机梯度下降算法用于迭代最优化均方误差代价函数

给定数据集 $X=\{x^{(1)},x^{(2)},\cdots,x^{(m)}\}$，数据集标记 $Y=\{y^{(1)},y^{(2)},\cdots,y^{(m)}\}$，
学习器 $f(x;w)=w_1x_1+w_2x_2+w_3x_3$，学习率（步长）$\alpha$。
For 迭代足够多次
{
随机选择一条数据 $x^{(j)}$；
$w_i=w_i+\alpha(y^{(j)}-f(x^{(j)};w))\cdot x_i^{(j)}$
}

算法优点很明显，那就是快，只要见到一条数据，就计算其梯度，然后调整参数 w。也许你会说缺点也很明显，那就是不可靠。但其真的“不可靠”吗？

我们先来聊聊 SGD 算法的优点。使用 SGD 算法，最大的好处就是我们不需要考虑数据集的尺寸，我们看见一条数据就修改一次参数。很自然地，我们的学习器就可以边学习边工作，也就是现在流行的**在线学习**（Online learning）[11]模式，更为重要的是社会是在发展变化的，同时数据也在不停地更新，就像 3 年前流行的穿衣风格，3 年后就可能彻底变了，而我们使用 3 年前最流行的穿衣风格数据来预测当前的穿衣风格，这就落伍了。

我们现在来谈谈 SGD 算法的“不可靠”，诚然，随机梯度下降使用的梯度是带有噪声的梯度，我们的下降方向也总是曲折的，但在这种曲折中，我们还是跌跌撞撞地游到了最小值附近，然后就在最小值周围震荡。虽然我们没有得到最优解，但我们至少得到了一个不错的解。并且在深度学习中，我们恰恰会避开最优解，而选择一个还不错的解，这其中有两个重要的原因。

其一就是我们的数据本身就不可靠，数据是带有噪声的，产生数据噪声的原因有很多，有人为造成的，也有天然的，**噪声是不可避免的**。为了解决这一问题，一个简单而又重要的想法呼之欲出，就是我们在噪声中学习。之前我们提到降噪任务就是在数据中加入噪声学习的一种学习方式，而随机梯度下降也可以认为是在梯度中加入噪声学习，这样训练出来的学习器就可能拥有更强的抗噪能力。

其二就是我们不要学得“太好”，这可能有点难以理解。在深度学习中，由于我们模型能力很强大，因此很容易就在已知数据中学得很“完美”，但在未知数据中就会表现得水土不服。深度学习中一个重要的方法就是使用**早停**（Early Stopping）[12]，也就是学到一定程度就终止了学习，这种方式没有在已知数据中表现得“完美”，反而在未知数据中表现得还不错。而随机梯度下降也没有在数据中学得“完美”，而在未知数据中，往往表现还不错。

当然如果噪声太大，我们也无法学习。取批量梯度下降与随机梯度下降的一个折中，如最早的算法 2.1 所示，我们称之为**最小批量梯度下降**（Mini-Batch Gradient Descent）[9]。假如全部数据有一百条，那我们可以随机选择 10 条数据求出其梯度误差，然后累加之后再取平均

值。但如何选择最小批量的大小，也是需要在实际应用中根据实际情况进行选择的。

2.4 过拟合与欠拟合

机器学习的核心任务是在新的、未知的数据中执行得好，而这种在未知数据中执行的能力，我们也称之为**泛化能力**（generalization）。

训练机器学习模型所使用的数据集称之为**训练集**（training set），而使用这些数据产生的误差称之为**训练错误**（training error）。在测试数据上的误差，称之为**测试错误**（test error）或**泛化错误**（generalization error），机器学习的目的就是去降低泛化错误。

但非常遗憾，虽然我们的目标是测试错误，但我们只能使用训练数据去训练我们的机器学习模型。那我们又凭什么去说服别人呢？统计学习理论给我们提供了些信心，如果训练数据与测试数据毫无关系，那我们就可以撕书离开了。但如果训练数据与测试数据（已知数据与未知数据）拥有某种关联性，我们就可以化腐朽为神奇了。

首先，假设训练数据和测试数据都是由某个概率分布生成的，我们将其称之为**数据生成过程**（data generating process），其实在介绍极大似然估计时也已经介绍过了，那就是数据**独立同分布**（independent and identically distributed，i.i.d.）假设。我们假设数据之间是相互独立的，而数据都是由某个概率分布函数生成，因此在训练数据（已知）上表现很好的算法，在测试数据集（未知）上依然也能表现得很好。

同时，这个假设也说明了我们如何做才能使机器学习算法执行得好，主要做到以下两点。

- 使训练错误率尽可能的低;
- 使训练错误率与测试错误率的差距尽可能的小。

这两个问题也引出了机器学习的两个核心问题：**欠拟合**（Underfitting）问题及**过拟合**（Overfitting）问题。当机器学习算法在训练数据上错误率较高时，我们就说这是欠拟合现象；当机器学习算法的测试错误率与训练错误率差距较大时，我们就说这是过度拟合现象。可能你对**拟合**（fit）一词还比较陌生，简单来说，我们把数据想象成空间中的点，然后找一条曲线穿过（或分割）这些点，这就叫作拟合。

过拟合与欠拟合现象，就像我们在过去事情上花费的精力，如果我们太过于计较细节，花很多精力去研究过往，那我们可能会深陷其中，反而对未来充满了畏惧；如果我们不在乎过去，不善于总结，那过去和未来都同样糟糕。生活讲究的就是一个度，如何去掌握度，不仅机器很难，其实对于生活更难。而文中所说的付出“精力”，对应到机器学习中就是算法的**能力或容量**（capacity）。模型的能力就是其拟合各种函数的能力，通俗来讲就是一个模型拥有多少种函数可以候选。模型能力低，那就很难去拟合训练数据集；模型能力高，则可能拟合训练数据集很好，但测试数据集就很差。通过调整机器学习算法的能力，我们需要寻找一个最佳的模型，既要在训练数据中表现很好，又要在测试数据中表现不错。

机器学习中控制模型能力的一种方式是选择其**假设空间**（Hypothesis Space），通俗些，就是控制算法可以使用的函数的数量。例如，我们使用线性回归去拟合数据，那线性回归包含的所有直线就是我们的假设空间。但想让直线穿过所有的点根本就不可能，那我们可以用二次函数去拟合数据，而且二次函数也包含了一次函数，更极端点，我们还可以用十次多项

式去拟合函数，此时的假设空间也就更大，算法能力也就更强了。

欠拟合　恰当能力　过拟合

(a)　(b)　(c)

图 2-3　不同假设空间拟合训练数据集示例

如图 2-3 所示，就说明了这一情况，图中的数据为二次函数生成的数据，在图 2-3（a）中我们使用了一次多项式去拟合数据，出现了欠拟合现象，而图 2-3（c）我们用九次多项式去拟合数据，虽然函数穿过了全部数据，但如果我们再添加新的数据进去，该函数就会出现较大的误差，因此就会发生过拟合现象。

我们需要记住的是，简单的函数更容易泛化，就像是一个愚笨的人虽然过去表现得不是很好，但未来也不会表现得太差；而复杂的函数往往在训练数据集上非常优异，但对于测试数据集，可能就会“聪明反被聪明误”。如图 2-4 所示，就是这种泛化能力与模型能力的“U 型曲线”，在最佳模型的左边，训练错误率与泛化错误率随着模型的能力的提高而下降；在最佳模型的右边，训练错误率随着模型能力的提升而下降，但泛化错误率却随着模型能力的提升而上升，最终，泛化错误率与训练错误率的差距越来越大。

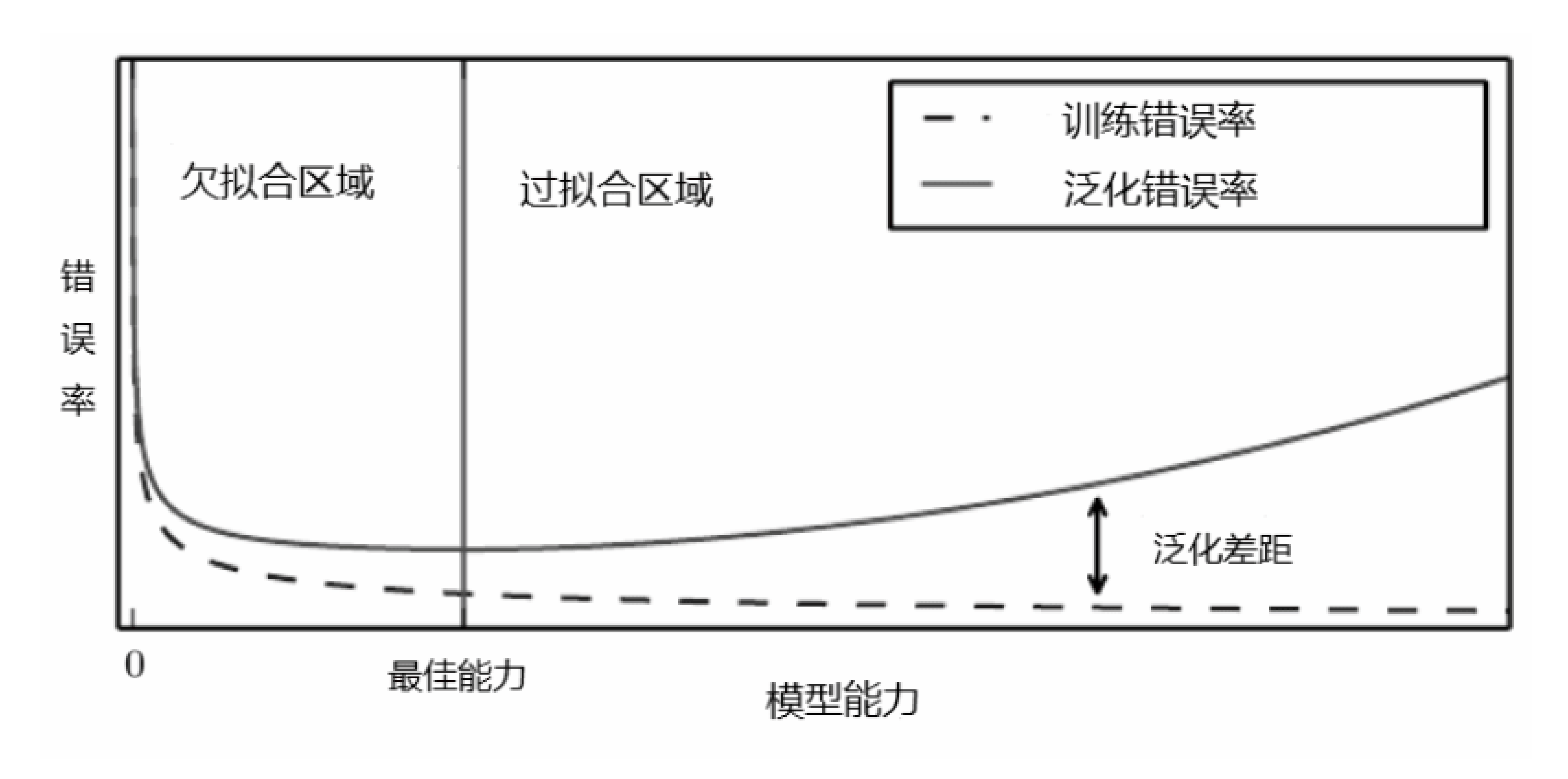


图 2-4　模型能力与错误率关系图

- 奥卡姆的剃刀

那我们实际中又如何选择各种不同能力的模型呢？“听过许多道理，依然过不好这一生”，可能就变成了我们最大的阻碍，现在引入一个古老而又简单的哲学思想，那就是**奥卡姆的剃刀**（Occam’s razor）。

“Do not multiply entities beyond necessity, but also do not reduce them beyond necessity.”

——William of Ockham

翻译过来就是“若无必要勿增实体，若无必要莫减实体”，通常前半段为奥卡姆的剃刀，但我觉得断章取义是对伟人的不敬，因此就全部写下了，仅供思考。这一思想应用在机器学习中就是，**如果两个假设空间都能很好地拟合数据，那就选择“最简单”的那一个**。虽然这一思想非常简单，但并不一定很容易做到，诸如大道至简的原理很多大家都提到过，接下来再引用爱因斯坦的一句至理名言，与这一思想类似，和大家一起瞻仰。

“Everything should be made as simple as possible, but not simpler.”

——Albert Einstein

2.4.1 没免费午餐理论

古往今来，不管是牛顿、爱因斯坦与霍金那样的巨人，还是无数默默耕耘的科学从业者，抑或是诸如我们这些仰望科学殿堂的游客，无不追寻或幻想着有一套理论，那就是**万物理论**。为了这一理论不知有多少人“衣带渐宽终不悔，为伊消得人憔悴”。而在机器学习界，也有许多人追寻或幻想着“超级智能算法”，但很遗憾，目前还不存在这样一个“超级智能算法”。1997年，Wolpert 就证明了这样一个理论，被称之为**没免费午餐理论**（No Free Lunch Theorem）[13]。该理论说明**所有可能的数据分布，所有分类算法在未知数据中都有着相同的错误率**。换而言之，就是没有一个通用的算法比其他算法好，这就如同非常复杂的概率图模型在所有任务中的平均性能和一个丢硬币算法（乱猜）的平均性能是相同的。其实就相当于你眼中的天才，让他去完成所有任务的性能，其实和一个普通人，甚至一个愚笨的人并没有区别。

不知你听完后会不会绝望，但如果你足够聪明的话，反而会看见更多的希望。注意该理论说的是**所有情况下的平均性能**，这也同时说明了，在特定任务中，特定的算法会比较优秀，这就好比“请不要让学编程的你去修电脑。”

同时也意味着，研究机器学习的目标不是去寻找一个通用的学习算法或者绝对的最好算法，相反，我们的目标是去寻找在我们关心的特定领域中的特定算法。

其实这也为平凡的我们找到了希望的大门，如果你现在不是很顺心，事业，学业不是很有希望。相信我，没关系的，总有一条属于你发光的路，总有一个领域你会比别人更优秀。

没有免费午餐理论指明，我们必须设计特定的机器学习算法，在特定的任务中执行。但什么样的算法适合在哪一种领域是要靠丰富的经验来进行选择的，不过请记住“经验”不是绝对的，若一种算法被普遍认为在某种任务中表现不好，但有一天“幸运女神”光顾了，你使用的不被看好的算法就会完成逆袭，那你也就可能从此走上“人生巅峰”，所谓科学精神，凝结成一点就是质疑精神。算法和数据是很有趣的，总有一个算法能找到“适合”它的数据，也总有数据能够找到“适合”它的算法，有时机器学习似乎不在讨论机器学习本身，讨论的好像是“运气”。不管是历史上还是在生活中，你是否思考过，许多将相名侯是命中注定的天选之子，还是上帝随意摇骰子选中的“幸运儿”。而我们谈论的人工智能，到底是“人工”，还是“智能”，当然这些问题是没有答案的，但多思考些没有答案的问题，你可能会得到一些不一样的快乐。

2.4.2　正则化

目前为止，我们修改机器学习算法的方法只是去增加或减小模型的能力。而这种调整，我们是通过增加或移除算法**可选择的假设空间**来实现的。假设我们的数据实际是由二次多项式函数生成的，但我们不是上帝，因此不可能知道真实的数据生成函数，而一个简单合理的方法就是我们从高次多项式到低次多项式逐个地尝试。

首先，我们使用九次多项式学习，那很有可能就会产生过拟合现象；然后使用八次多项式学习，再测试性能；最后我们通过这种锲而不舍的精神，一次次地尝试，测试到二次多项式时，其方法最理想，那我们就确定了学习算法的假设空间。这种方法先不提时间问题，每次都要重新配置学习算法就是一个很大的工程量，那我们可不可以稍微修正一下我们从高次到低次逐渐尝试的耿直方法呢？

我们从高次到低次尝试的过程其实就是在限制参数的个数，如式（2.16）所示的九次多项式。

$$f(x) = w_1x^9 + w_2x^8 + \cdots + w_8x^2 + w_9x^1 + w_{10}x^0 \tag{2.16}$$

式（2.17）是二次多项式，是测试中的最佳公式。

$$f(x) = 0x^9 + 0x^8 + \cdots + 0x^3 + w_8x^2 + w_9x^1 + w_{10}x^0 \tag{2.17}$$

从式（2.17）中可以看出，其实就是高次项的系数为零。

而我们放松这一苛刻限制的方式，如式（2.18）所示，将 0 系数改为较小的系数。

$$f(x) = 0.0001x^9 + 0.0003x^8 + \cdots + 0.002x^3 + w_8x^2 + w_9x^1 + w_{10}x^0 \tag{2.18}$$

那如何自动地去调整这些参数呢？其实很简单，只需要在代价函数中，加入一项参数惩罚即可。如式（2.19）所示，将**权重衰减**（Weight Decay）[14]加入到均方误差中。

$$J(\boldsymbol{w}) = (y - f(x;\boldsymbol{w}))^2 + \lambda \boldsymbol{w}^{\mathrm{T}}\boldsymbol{w} \tag{2.19}$$

其中参数 $\boldsymbol{w}$ 是用向量表示的，如果不熟悉线性代数，简单地理解成对应参数的平方求和即可，如果数据特征维度仅为一维，也可以简单转化成式（2.20）的样式。

$$J(w) = (y - f(x;w))^2 + \lambda w^2 \tag{2.20}$$

关于上述的参数惩罚你可能会一脸茫然，但别急，我们将会在第 4 章详细地介绍深度学习中各种重要的参数正则化方法，那会非常精彩，而现在就假装你已经知道这一方法的原理即可。

式（2.20）中的 λ 用于控制权重的稀疏性。当 $\lambda=0$ 时，就没有任何权重惩罚，如果 λ 变大就会迫使模型权重（参数）变小。如图 2-5 所示，显示了调整 λ 值对于模型能力的影响，其中图 2-5（c）为 λ 趋近于 0，模型退化成了九次多项式函数，由此造成了过拟合现象；而图 2-5（a）由于 λ 的值过大，模型退化成了一次函数，导致了欠拟合现象；而图 2-5（b）为选择了适当的 λ 值，拟合到了正确曲线。

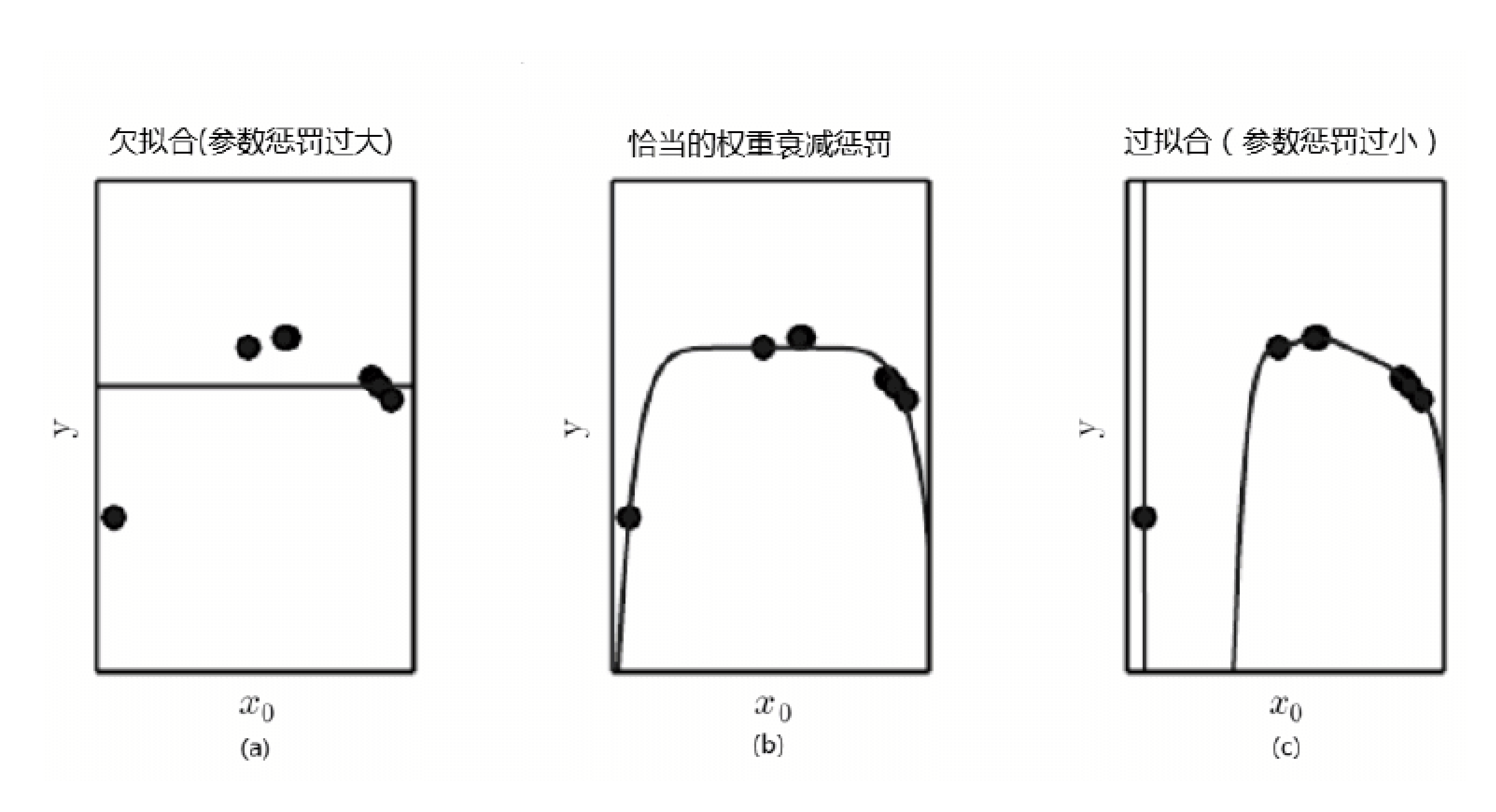


图 2-5　权重衰减对于模型能力影响示意图

正式地来说，所谓**正则化**（Regularization）就是想要降低泛化错误率但不降低训练错误率而修改机器学习算法的一系列方法。没免费午餐理论已经说明了没有最好的机器学习算法，同样地，机器学习中也没有最好的正则化方式。因此，选择适合的特定任务，特定算法的正则化方式也需要进行大量的实验。

2.5　超参数与验证集

我们已经学习了不少机器学习的概念，同时我们也引入了不少符号及专业术语。不知你是否已经被这些概念弄糊涂了，现在我们就对这些内容进行梳理。我们可以很粗浅地认为，机器学习其实就是通过一些优化手段去调整数据权重 w（参数）。我们也了解到了梯度下降中的学习率 α（步长）和正则化中权重衰减的惩罚因子 λ 都对机器学习算法的最终性能产生着巨大的影响，在学习过程中我们需要不停地调整学习率与惩罚因子，那么如何去区分这些不同的符号呢?

我们目前说到的学习，通俗点也可以说是去寻找最佳的权重 w，而学习率与惩罚因子这些设置，其实就是帮助我们去寻找最佳的权重，这些设置控制着机器学习算法的行为，我们就将其统称为**超参数**（Hyperparameters）。

虽然超参数与参数我们都需要去调整，但通常针对参数的调整叫作学习，而对超参数的调整叫作选择。其原因在于，我们通常在训练数据集中是不去修改超参数的。接下来引入一个例子帮助你理解参数和超参数，“杨老师想在班上选择一个‘爬树小冠军’，但大家都不会爬树，不过小灿、小嫣和小姣同学热情都很高涨。那杨老师就指着一棵树说，你们先学会儿爬树吧，然后再选出最厉害的那个。于是小嫣同学就去学了半小时的爬树技能，结果摔得鼻青脸肿，不过至少还是学会了爬树；小姣同学就比较灵敏学得快，爬得也快；小灿同学就不得了，经过半小时的学习就和小猴子一样厉害了。老师最后就让他们去爬另外一棵树，最终，小灿获得了‘猴王’称号。”

以上例子中的同学各自学习爬树的过程就相当于机器学习中的学习过程，需要注意的是，她们都是学习爬同一棵树，也就是都是在训练数据集中学习。而老师选择哪一个厉害，就相当于超参数选择，选择是在一棵新树上进行测试的。这里需要注意的一个小知识点是，超参数选择的是同一算法的不同性能，或者是对同一算法族的选择。就好像小灿、小嫣和小姣都

是人，而且还都是女生。如果是对小猴、小猫和小狗的选择，那就不算是超参数选择了，那是不同算法的比较了。

以上例子中还有一个重点在于老师是让她们在一棵“新树”进行比赛的，需要特别注意的是这里的“新树”不是测试数据。刚开始进入机器学习的“青椒”们常见的误区就在于此，我们这里所说的“新树”，称之为**验证数据集**（validation set）。

简单来说，我们将已知数据分成两大部分，一部分用于训练，一部分用于测试。而在训练数据集中我们又可分为两部分，一部分用于学习参数，我们称为**训练数据集**；一部分用于选择超参数，我们称之为**验证数据集**。而最终性能测试的数据，我们称为**测试数据集**。

那我们为什么要这样划分呢？

我们先来玩一个简单的摇骰子游戏，有三个骰子，每个骰子有 6 个面，如果能同时摇出三个 6，那就可以得到一颗棒棒糖，奖励给我们的“天选之子”。

正常情况下，摇出三个 6 的概率可看做独立重复事件，其概率为：

$$P(\text{“摇出豹子的概率”}) = \left(\frac{1}{6}\right)^3 = \frac{1}{216} \approx 0.0046$$

我们能摇出“豹子”的概率只有 0.46%，就连百分之一都没有，假设我们班共有 100 个同学玩这个游戏，那么至少有一人获得棒棒糖的概率又为多少呢？

$$P(\text{“整个班级至少一人获奖”}) = 1 - \left(\frac{215}{216}\right)^{100} \approx 0.371$$

我们班里至少一个同学摇出“豹子”的概率为 0.371，看来棒棒糖送出去的概率还是比较小的。可是当我们这个游戏被升级为“年级摇骰王选拔赛”时，我们共有 7 个班，每个班都可以参与进来。假设每个班人数相同，那整个年级至少一人摇出豹子的概率又为多少呢？

$$P(\text{“整个年级至少一人获奖”}) = 1 - \left(\frac{215}{216}\right)^{700} \approx 0.961$$

我们整个年级能摇出“豹子”的概率就高达 0.961 了，这几乎可以断定，一定会出现这么一个“天选之人”了。

介绍完了以上的小游戏，不知你会不会有点失落。机器学习算法可以想象成在空间中找“分割面”，机器学习算法能力的大小也可以说成是可选分割面数量的多少。机器学习从某种程度上而言不就是去寻找我们游戏中的“豹子”吗？一个人要摇出“豹子”的概率是很低的，但让 7 个班、每班 100 个人去摇骰子，出现“豹子”的概率就非常大了。我们天真地以为这个“天选之人”就是赌神，但其实他和我们一样普通。

一个班级就相当于一套超参数配置下的学习算法，配置数量越多，就相当于参加比赛的班级越多。我们在所有超参数配置中找一个最佳性能参数，不就是相当于在所有班级中找一个“摇骰王”吗？

我们的模型能力越大，那最终在训练数据集中的表现也就会越好，但这个“好”是乱猜还是学习，就需要在新的数据上测试一番，但我们并没有真正的新数据，我们只能划分一部分训练数据去充当“新数据”。初学者可能会把测试数据当成验证数据来用，但要记住，**验证数据是帮你去选择超参数的**，虽然验证数据不直接参与训练过程，但**进行超参数选择的时候，其实也间接地包含在整个学习过程中**。

那怎样去划分训练数据与验证数据呢?

通常使用 80%的训练数据作为训练，20%的数据用于验证。由于验证数据也用于“训练”，因此验证集的错误率是要低于真实的泛化错误率的。当配置出了最佳的超参数时，再用测试数据集去测试学习器的泛化性能。

通常将数据集分成固定的训练集与固定的验证集，使用训练数据集去学习，使用验证数据集去验证训练的结果。但测试是存在偶然性的，也许划分出来的训练数据与验证数据差别很大，可能我们的机器学习算法性能也挺不错的，但由于验证数据的问题，而变成了“窦娥冤”；也许我们的算法其实一般，但恰恰就很幸运地在验证数据集上表现优异。为了减少这种偶然性，使用**平均验证错误**是一个比较好的方式，而 **K 折交叉验证**（K-fold Cross-Validation）[15]就是其最常用的一个方法。

K 折交叉验证先将数据集分割成大小相同的 K 组数据集，我们先使用 2~K 组数据集作为训练数据集进行训练，而后用第一组数据集进行验证；接下来使用第二组数据集作为验证数据，而剩余的作为训练数据，直至遍历完所有数据集，将 K 组验证错误率取平均值，而平均验证错误率就当作泛化错误率。

K 最常用的取值为 10，称为 10 折交叉验证。K 的取值越大，划分的数据集越多，那最终的泛化错误率的可靠性就越高，但相应的时间花费就越大。因此 K 的取值需要在实际应用中，在训练时间与可靠性中做一个取舍，其也是一对鱼和熊掌。

2.6　Softmax 编码实战

讲了太多的理论知识，接下来我们将动手实现第一个学习模型——Softmax[16]多分类器。该分类器是深度学习中网络输出层的默认分类器，因此也可以认为是**最简单**的多分类“神经网络”。首先我们会介绍该模型的一些理论知识，然后你需要使用该模型并结合上述我们介绍的机器学习知识，完整地完成 CIFAR-10 分类图像识别任务。

假如让你来设计一个分类互斥的三分类函数你会怎么设计呢？你可能一开始会有点不知所措，但记住，当我们想要创造点什么的时候，我们首先要思考的其实是我们拥有着什么。

可以把三分类任务当作是三个人来处理，特定的人就代表着擅长于判断特定的类别。那每个人可以做些什么呢？其实很简单，就如同线性回归一样，每个人只是对数据特征进行加权求和。如式（2.21）所示，将数据维度乘以权重再求和，也将其称为**评分函数**（score function）。

$$z=\sum_{i=0}^{m} w_i x_i \qquad \text{（式 2.21）}$$

z 就表示每个人心中的分类得分，有三个人，也就会有三个 z 值，选取其中最高分，也就是“最自信”的分值即可。

那这就有一个问题，比如一个识别小猫的任务，得分可能是(50,40,45)，也可能是(25,1,2)。那么请问哪一组评分系统好？虽然两组中第一列都是最高得分，但明显后者的相对得分更为突出，那就相当于第二组中的第一列，比其他列有更高的自信心。因此除了关注最高分以外，还要关心其相对比值，也就是**将各自得分再除以总分**即可。

如式（2.22）所示，就是我们设计的分类公式。非常简单，我们仅仅将所有人的分数之和作为公分母，然后将每个人各自的分数作为分子，最后我们就得到了总和为 1 的多分类函

数，这一步也叫**归一化概率**（Normalized Probabilities）。

$$f(x)=\frac{1}{z^1+z^2+z^3}\begin{bmatrix} z^1 \\ z^2 \\ z^3 \end{bmatrix} \tag{2.22}$$

我们将式（2.21）与式（2.22）合并一下，得到式（2.23），就是其完整的表达式，虽然看起来比较复杂，但是所表达含义却非常简单。

$$f(x)=\frac{1}{\sum_{j=1}^{3}\sum_{i=0}^{m} w_i^j x_i^j}\begin{bmatrix} \sum_{i=0}^{m} w_i^1 x_i^1 \\ \sum_{i=0}^{m} w_i^2 x_i^2 \\ \sum_{i=0}^{m} w_i^3 x_i^3 \end{bmatrix} \quad \text{（式 2.23）}$$

上述表达式还缺点东西，因为我们想要预测得分比尽可能的高，但得分函数是线性的，其增长幅度有限。那可不可以在不影响其单调性的情况下，使其相对得分更显著呢？如式（2.24）所示，这就是我们修改之后的表达式，由于指数函数是一个单调函数，因此这样的修改并不会对表达式含义有任何影响。

$$z=\mathrm{e}^{\sum_{i=0}^{m} w_i x_i} \tag{2.24}$$

相应地，我们将这些改变全部写在一起，就是式（2.25）。

$$f(x)=\frac{1}{\sum_{j=1}^{3} \mathrm{e}_j^{\sum_{i=0}^{m} w_i x_i}}\begin{bmatrix} \mathrm{e}_1^{\sum_{i=0}^{m} w_i x_i} \\ \mathrm{e}_2^{\sum_{i=0}^{m} w_i x_i} \\ \mathrm{e}_3^{\sum_{i=0}^{m} w_i x_i} \end{bmatrix} \tag{2.25}$$

我们将其推广至 k 类，如式（2.26）所示，这就是 Softmax 函数表达式。

$$f(x)=\frac{1}{\sum_{j=1}^{k} \mathrm{e}_j^{\sum_{i=0}^{m} w_i x_i}}\begin{bmatrix} \mathrm{e}_1^{\sum_{i=0}^{m} w_i x_i} \\ \mathrm{e}_2^{\sum_{i=0}^{m} w_i x_i} \\ \vdots \\ \mathrm{e}_k^{\sum_{i=0}^{m} w_i x_i} \end{bmatrix} \tag{2.26}$$

第一眼看见该表达式，可能会有点抓狂，但其实这并不复杂。我们之所以花那么多“无用”的时间来推导出该公式，只是想告诉你，只要你不再害怕，不再抗拒时，静下心来你就会发现这有多简单。

Softmax 函数的输出是一个 k 维向量，比如函数输出为[0.1,0.3,0.7]，而该数据的真实标记为第二类，其向量表示就应该是[0,1,0]。那我们如何去刻画其误差呢？你也许会想到将各自的

误差加起来，比如使用均方误差作为代价函数，那么该数据的误差就应该如下所示。

$$J(x) = (0.1)^2 + (0.3-1)^2 + (0.7)^2$$

但实际上我们多算了一些误差，因为这三个输出是彼此依赖的，调整其中一个，其余都会受到影响。因此正确的代价误差应该如式（2.27）所示。

$$J(x) = (0.3-1)^2 \tag{2.27}$$

虽然 Softmax 的输出为 k 个输出，但我们仅对其真正关键的那一个输出进行修改就可以。为了简化接下来的描述，我们将引入如式（2.28）所示的示例函数。

$$I\{表达式\} = \begin{cases} 1，表达式为真 \\ 0，表达式为假 \end{cases} \tag{2.28}$$

其使用方法非常简单，表达式的输出为真则输出 1，表达式的输出为假则输出 0，例如：$I\{2+3=4\}=0$，$I\{1+1=2\}=1$。

Softmax 代价函数就可表示为式（2.29），其中 y 表示数据的真实类标，比如 $y=3$，就表示该数据为第三类数据，该代价函数所表达的含义其实就为遍历所有输出，将真实类标对应的输出的误差，作为我们的代价函数。

$$J(w) = -(\sum_{j=1}^{k} I\{y=j\}\ln\frac{\mathrm{e}_j^{\sum_{i=0}^{m} w_i x_i}}{\sum_{l=1}^{k}\mathrm{e}_l^{\sum_{i=0}^{m} w_i x_i}}) \tag{2.29}$$

对我们来说，更为实际的可能是梯度的计算公式，因为使用代价函数推导梯度计算公式后，我们才能根据其修改权重。这里不进行 Softmax 的梯度推导，直接给出梯度计算公式，如式（2.30）所示。

$$\nabla w_i^{(j)} = -x_i(I\{y=j\} - p_j(x)) \tag{2.30}$$

其中，$p_j(x) = \frac{\mathrm{e}_j^{\sum_{i=0}^{m} w_i x_i}}{\sum_{l=1}^{k}\mathrm{e}_l^{\sum_{i=0}^{m} w_i x_i}}$，若你不太习惯式（2.30）的写法，可以写成式（2.31），可能更有助于记忆。

$$\nabla w_i^{(j)} = \begin{cases} -x_i(1-p_j(x))，\ y=j \\ x_i p_j(x)，\ y \neq j \end{cases} \tag{2.31}$$

这其实就是交叉熵代价函数在多类任务的推广，再将式（2.31）换一种写法就得到式（2.32）。

$$\nabla w_i = \begin{cases} -x_i(1-f(x))，\ y=1 \\ x_i f(x)，\ y \neq 1 \end{cases} \tag{2.32}$$

这就是典型的二分类交叉熵梯度计算公式。

- Softmax 参数冗余

Softmax 存在着一个问题，那就是参数冗余。假设我们进行二分类任务，Softmax 的输出为一个二维向量，就相当于有着两套数据参数，在逻辑回归中，预测生病的概率为 0.7，那不生病的概率很自然就是 0.3。而使用 Softmax 的方法使我们计算出生病的概率为 0.7，但同样也计算出了不生病的概率为 0.3。因此 Softmax 神经元总是比实际需要多了一套参数，但这相比于神经网络的上亿参数，简直微乎其微，这里仅作为冷门知识了解即可。

2.6.1　编码说明

在本章的练习中我们将要逐步完成以下内容。

1. 熟悉使用 CIFAR-10 数据集；
2. 编码 softmax_loss_naive 函数，使用显式循环计算损失函数以及梯度；
3. 编码 softmax_loss_vectorized 函数，使用向量化表达式计算损失函数及其梯度；
4. 编码随机梯度下降算法，训练一个 Softmax 分类器；
5. 使用验证数据选择超参数。

完整的教程请参考“第 2 章练习-实现 softmax.ipynb”文件顺序练习，该处仅仅对重要内容进行解释说明。在本章结尾将会提供参考代码，希望你“走投无路”时再查看这些“锦囊”。请记住“Practice makes perfect”。

首先启动 Jupyter：如图 2-6 所示，启动 dos 窗口，将路径转换到文件：“第 2 章练习-实现 softmax.ipynb”所在路径，然后输入：jupyter notebook，再按 Enter 键确认。这时，浏览器会自动启动 Jupyer，单击本章练习即可。

```
管理员: 命令提示符 - jupyter notebook
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation。保留所有权利。

C:\WINDOWS\system32>cd f:\DLAction

C:\WINDOWS\system32>f:

f:\DLAction>jupyter notebook
[I 19:16:23.976 NotebookApp] [nb_conda_kernels] enabled, 2 kernels found
[I 19:16:28.764 NotebookApp] [nb_anacondacloud] enabled
[I 19:16:29.379 NotebookApp] \u2713 nbpresent HTML export ENABLED
[W 19:16:29.379 NotebookApp] \u2717 nbpresent PDF export DISABLED: No module named 'nbbrowserpdf'
[I 19:16:29.464 NotebookApp] [nb_conda] enabled
[I 19:16:29.701 NotebookApp] Serving notebooks from local directory: f:\DLAction
[I 19:16:29.701 NotebookApp] 0 active kernels
[I 19:16:29.701 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
[I 19:16:29.701 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
```

图 2-6　启动 Jupyter

首先，我们需要导入各模块，由于 Python 2.7+系列在默认情况下不支持中文字符，因此，你需要在每个文件的开头添加一行“#-*- coding: utf-8 -*-”，才能使用中文注释，否则编译器将会抛出编码异常错误。本章所需完成的代码模块全部存放在文件“DLAction/classifiers/chapter2”中，而诸如数据导入和梯度检验等模块被统一存放在了“DLAction/utils”目录下，读者可自行查看。

库导入代码块：

```
#-*- coding: utf-8 -*-
import random
import numpy as np
from utils.data_utils import load_CIFAR10
from classifiers.chapter2 import *
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams[ 'figure.figsize' ] = ( 10.0, 8.0 )
%load_ext autoreload
%autoreload 2
```

2.6.2 熟练使用 CIFAR-10 数据集

CIFAR-10 是一套包含了 60000 张，大小为 32×32 的十分类图片数据集，其中 50000 张图片被分为了训练数据，10000 张图片被分为测试数据，存放在“DLAction/datasets/cifar-10-batches-py”目录下，也可以通过访问互联网地址进行下载 http://www.cs.toronto.edu/~kriz/cifar.html。由于我们使用的是彩色图片，因此每张图片都会有三个色道。当我们将磁盘图片导入到内存时，该数据集存放在形状如下的 NumPy 数组中。

CIFAR-10 数据格式：

```
训练数据（数据个数，数据维度）：(50000L, 32L, 32L, 3L)
训练数据标记（数据标记个数，）：(50000L,)
测试数据（数据个数，数据维度）：(10000L, 32L, 32L, 3L)
测试数据标记（数据标记个数，）：(10000L,)
```

载入 CIFAR-10 数据的各项操作已经被封装进了 load_CIFAR10 函数中，只需要载入数据的存放目录，就可以得到划分好的训练数据、训练数据类标、测试数据和测试数据类标。

导入 CIFAR-10 数据代码块：

```
# 导入 CIFAR-10 数据。
cifar10_dir = 'datasets/cifar-10-batches-py'
X_train, y_train, X_test, y_test = load_CIFAR10( cifar10_dir )
# 查看数据。
print '训练数据（数据个数，数据维度）: ', X_train.shape
print '训练数据标记（数据标记个数，）: ', y_train.shape
print '测试数据（数据个数，数据维度）: ', X_test.shape
print '测试数据标记（数据标记个数，）: ', y_test.shape
```

如果觉得这些数据还是有些抽象，你也可以使用下列的代码块，可视化地查看载入 CIFAR-10 数据集图片。

CIFAR-10 数据可视化代码块：

```
# 数据可视化.
classes = [ 'plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck' ]
num_classes = len( classes )
samples_per_class = 7
for y, cls in enumerate( classes ):
    idxs = np.flatnonzero( y_train == y )
    idxs = np.random.choice( idxs, samples_per_class, replace = False )
    for i, idx in enumerate( idxs ):
        plt_idx = i * num_classes + y + 1
        plt.subplot( samples_per_class, num_classes, plt_idx )
        plt.imshow( X_train[ idx ].astype( 'uint8' ) )
        plt.axis( 'off' )
        if i == 0:
            plt.title( cls )
plt.show( )
```

其图像如图 2-7 所示。



图 2-7　CIFAR-10 数据集

- 数据预处理

数据划分：原始数据量可能稍大，这样不利于我们的编码测试，因此我们会采样 250 条训练数据作为样本训练数据 X_sample，采样 100 条数据作为样本验证数据 X_validation，作为编码阶段测试使用。

数据形状转换：原始的数据集可看成是四维数组(数据个数，宽，高，色道)，这样不太方便使用，因此我们将(宽，高，色道)压缩在一维上，其数据形式变为(数据个数，数据维度)，而数据维度=宽×高×色道。数据形状转换代码如下所示。

数据形状转换代码块：

```
X_train = np.reshape( X_train, ( X_train.shape[ 0 ], -1 ) )
X_val = np.reshape( X_val, ( X_val.shape[ 0 ], -1 ) )
X_test = np.reshape( X_test, ( X_test.shape[ 0 ], -1 ) )
X_sample = np.reshape( X_sample, ( X_sample.shape[ 0 ], -1 ) )
```

数据归一化（Normalization）：在通常情况下，我们需要对输入数据进行归一化处理，也就是使得数据呈**均值为零，方差为 1 的标准正态分布**。由于图像的特征范围在[0,255]，其方差已经被约束了，我们只需要将数据进行**零均值中心化**处理即可，不需要将数据压缩在[-1,1]范围（当然，也可以进行此项处理）。因此在处理时，只需要减去其数据均值即可。**注意：**我们计算的是训练数据的均值，而不是全部数据的均值，你需要时刻警惕不要“偷看期末试卷”，测试数据是“未知的”。数据归一化实现代码如下所示。

图像数据归一化代码块：

```
mean_image = np.mean( X_train, axis = 0 )
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_sample -= mean_image
```

添加偏置项：偏置项（bias）也可以看作是线性函数的常数项或截距，在实际中并不特别地区分偏置项参数与权重参数，并且其对于训练效果的影响也非常有限。但为了遵照传统，我们还是将其默认地使用在算法中，通过在数据中增加一维值为常数 1 的特征作为 bias 对应的输入特征，然后将其统一到权重参数中，最终的图片数据维度就为 dim=32×32×3+1。其实现的代码如下所示。

数据中加入偏置项代码块：

```
X_train = np.hstack( [ X_train, np.ones( ( X_train.shape[ 0 ], 1 ) ) ] )
X_val = np.hstack( [ X_val, np.ones( ( X_val.shape[ 0 ], 1 ) ) ] )
X_test = np.hstack( [ X_test, np.ones( ( X_test.shape[ 0 ], 1 ) ) ] )
X_sample = np.hstack( [ X_sample, np.ones( ( X_sample.shape[ 0 ], 1 ) ) ] )
```

我们将这些步骤统统封装在 get_CIFAR10_data()函数中。运行该函数，将得到以下结果。

```
X_train, y_train, X_val, y_val, X_test, y_test, X_sample, y_sample = get_CIFAR10_data( )
Train data shape:   (49000L, 3073L)
Train labels shape:   (49000L,)
Validation data shape:   (100L, 3073L)
```

```
Validation labels shape:   (100L,)
Test data shape:   (10000L, 3073L)
Test labels shape:   (10000L,)
sample data shape:   (250L, 3073L)
sample labels shape:   (250L,)
```

2.6.3 显式循环计算损失函数及其梯度

首先使用循环的方式实现 Softmax 分类器的损失函数（代价函数），打开“DLAction/classifiers/chapter2/softmax_loss.py”文件并实现 softmax_loss_naive 函数。

Softmax 损失函数伪代码:
1.从训练数据中采样一小批数据；
2.计算每条数据对应的各得分函数值；
3.计算每条数据得分函数的指数得分；
4.计算每条数据的总得分；
5.将各类得分除以总得分，计算每条数据的分类概率；
6.计算每条数据的损失值；
7.计算每条数据产生的梯度值；
8.将 6 中的各条数据损失值累加起来，计算平均损失值；
9.将 8 中的平均损失值加上权重衰减损失值；
10.将 7 中各条数据产生的梯度值累加起来，计算平均梯度值；
11.将 10 中的平均梯度值再加上权重衰减梯度。

注意：请尽量少地使用循环，使用的循环越少，就可以节约更多的计算时间。需要逐渐地适应向量化计算表达，因为采用向量化表达，既书写简洁，使得代码的可读性大大提高，不容易产生错误，又极大地提高了运算效率。例如计算每类得分的指数，就可以直接使用 scores_E=np.exp(X[i].dot(W))函数，更多 NumPy 的用法请参考第 1 章 1.3 Python 简易教程。

softmax_loss_naive 代码块:

```
def softmax_loss_naive(W, X, y, reg):
    """
    使用显式循环版本计算 Softmax 损失函数。
    N 表示：数据个数，D 表示：数据维度，C：表示数据类别个数。
    Inputs:
    - W: 形状( D, C ) numpy 数组，表示分类器权重（参数）。
    - X: 形状( N, D ) numpy 数组，表示训练数据。
    - y: 形状( N, ) numpy 数组，表示数据类标，
        其中 y[ i ] = c 意味着 X[ i ]为第 c 类数据，c 取值为[ 0, c )。
    - reg: 正则化惩罚系数。
```

```
Returns  二元组( tuple ):
- loss：数据损失值。
- dW：权重 W 所对应的梯度，其形状和 W 相同。
"""
# 初始化损失值与梯度。
loss = 0.0
dW = np.zeros_like( W )
#############################################################################
#  任务：使用显式循环实现 softmax 损失值 loss 及相应的梯度 dW 。          #
#  提示：如果不慎，将很容易造成数值上溢，别忘了正则化。                     #
#############################################################################




#############################################################################
#                               结束编码                                    #
#############################################################################
return loss, dW
```

完成了上述代码后，要怎样判断实现是否正确？一个比较直接的方法就是计算其损失值。在不加入权重惩罚的情况下，所实现的 Softmax 损失值应该接近于-log(0.1)。为什么是 0.1？如果是 5 分类任务，该损失值又该接近于多少呢？损失值验证代码块如下所示。

损失值验证代码块：

```
from classifiers.chapter2.softmax_loss import softmax_loss_naive
import time
# 初始化权重。
W = np.random.randn( 3073, 10 ) * 0.0001
loss, grad = softmax_loss_naive( W, X_sample, y_sample, 0.0 )
# 你的初始化损失值应该接近于 -log( 0.1 )。
print '你实现的 softmax 损失值 loss: %f' % loss
print '正确的损失值: %f' % ( -np.log( 0.1 ) )
```

其正确结果近似于这样。

损失值验证代码执行结果：

```
你实现的 softmax 损失值 loss: 2.329645
正确的损失值: 2.302585
```

接下来我们进行梯度检验，精确的数值梯度是使用极限的方式求解梯度，如式（2.33）所示。

$$\nabla w = \lim \frac{J(w+\varepsilon)-J(w-\varepsilon)}{2\varepsilon} \tag{2.33}$$

数值梯度的优势是比较精确，但缺点也很明显，那就是速度较慢。因此我们通常求解代价函数的导函数来替换数值梯度，但导函数在人工实现时很容易出错，我们已经实现了以极限方式的数值梯度求解，那么可以使用该函数检验实现的梯度函数。运行下列代码，相对误差应该小于 1e-7。

梯度验证代码块：

```
# 使用数值梯度检验已实现的 softmax_loss_naive。
# 实现的梯度应该要接近于数值梯度。
from utils.gradient_check import grad_check_sparse
loss, grad = softmax_loss_naive( W, X_sample, y_sample, 0.0 )
print '检验无权重衰减的 softmax_loss_naive 梯度:'
f = lambda w: softmax_loss_naive( w, X_sample, y_sample, 0.0 )[ 0 ]
grad_numerical = grad_check_sparse( f, W, grad, 10 )
print '检验加入权重衰减项后的 softmax_loss_naive 梯度:'
loss, grad = softmax_loss_naive( W, X_sample, y_sample, 1e2 )
f = lambda w: softmax_loss_naive( w, X_sample, y_sample, 1e2 )[ 0 ]
grad_numerical = grad_check_sparse( f, W, grad, 10 )
```

其正确的结果应该如下所示。

softmax_loss_naive 函数梯度检验结果：

```
检验无权重衰减的 softmax_loss_naive 梯度:
numerical: 1.862868 analytic: 1.862868, relative error: 2.737970e-08
numerical: -0.456766 analytic: -0.456766, relative error: 4.118098e-08
numerical: -1.964069 analytic: -1.964069, relative error: 1.885641e-08
numerical: 0.293490 analytic: 0.293490, relative error: 1.564932e-08
numerical: 0.370430 analytic: 0.370430, relative error: 1.449973e-07
numerical: -1.669150 analytic: -1.669150, relative error: 3.227082e-08
numerical: -3.304948 analytic: -3.304948, relative error: 1.310918e-08
numerical: -1.348797 analytic: -1.348797, relative error: 3.011066e-08
numerical: -1.251505 analytic: -1.251505, relative error: 8.791012e-08
numerical: -4.628194 analytic: -4.628194, relative error: 4.320206e-10
检验加入权重衰减项后的 softmax_loss_naive 梯度:
numerical: -0.714441 analytic: -0.714441, relative error: 2.873075e-08
numerical: 1.076180 analytic: 1.076180, relative error: 5.406905e-08
numerical: -0.478508 analytic: -0.478508, relative error: 1.251312e-08
numerical: 0.281967 analytic: 0.281967, relative error: 4.291053e-08
numerical: -2.360461 analytic: -2.360461, relative error: 1.525908e-10
```

```
numerical: 2.422492 analytic: 2.422491, relative error: 1.830325e-08
numerical: -4.721960 analytic: -4.721960, relative error: 1.496757e-08
numerical: 2.231498 analytic: 2.231497, relative error: 4.564722e-08
numerical: -1.063935 analytic: -1.063935, relative error: 6.914466e-08
numerical: 0.784813 analytic: 0.784813, relative error: 4.133483e-08
```

2.6.4 向量化表达式计算损失函数及其梯度

完成显式循环计算后，我们将损失函数与梯度的计算过程，使用完全向量形式再完成一遍。向量形式不仅书写简洁，并且也极大地加快了执行效率，对于编程人员来说，一开始使用向量化表达可能十分痛苦，但当慢慢熟悉后会对其着迷。打开“DLAction/classifiers/chapter2/softmax_loss.py”文件，文件实现了 softmax_loss_vectorized 函数。可以参考第 1 章中介绍的 NumPy 广播的用法，不到山穷水尽，请不要偷看参考代码。

提示：比如计算得分时，可以一次性求解所有训练数据 scores=np.dot(X,W)，此时 scores 变成形状为（数据个数，分类个数）的矩阵。输入参数 *y* 为一个类标向量(数组)，若 *y*[*i*]=2 就表示第 *i* 条数据的正确分类为 2，但 Softmax 分类器生成的分类得分概率为一个矩阵(二维数组)。因此，需要将类标向量 *y* 转换为 one-hot（向量中类标位为 1，其余为零），比如 *y*[*i*]=2 的类标，转化为 one-hot 就为[0,0,1,0,0,0,0,0,0,0]。我们可以使用以下代码进行 one-hot 形式的类标矩阵转换：y_trueClass[range(num_train),y]=1.0，其形状为(数据个数，分类个数)，这样就可以在后续的计算中使用向量计算。

softmax_loss_vectorized 代码块：

```
def softmax_loss_vectorized( W, X, y, reg ):
    """
    Softmax 损失函数，使用矢量计算版本。
    输入、输出格式与 softmax_loss_naive 相同。
    """
    # 初始化损失值与梯度。
    loss = 0.0
    dW = np.zeros_like( W )
    #################################################################
    # 任务：不使用显式循环计算 softmax 的损失值 loss 及其梯度 dW。          #
    # 提示：如果不慎，将很容易造成数值上溢。别忘了正则化。                   #
    #################################################################



    #################################################################
    #                          结束编码                              #
    #################################################################
    return loss, dW
```

接下来，使用前面已实现的 softmax_loss_naive 与向量化版本进行比较，完全向量化版本应该和显式循环版本的结果相同，但前者的计算效率应该快得很多。运行下列代码进行代码检验。

softmax_loss_vectorized 检验代码块：

```
tic = time.time( )
loss_naive, grad_naive = softmax_loss_naive( W, X_sample, y_sample, 0.00001 )
toc = time.time( )
print '显式循环版 loss: %e     花费时间  %fs' % ( loss_naive, toc – tic )
from classifiers.chapter2.softmax_loss import softmax_loss_vectorized
tic = time.time( )
loss_vectorized, grad_vectorized = softmax_loss_vectorized( W, X_sample, y_sample, 0.00001 )
toc = time.time( )
print '向量化版本  loss: %e      花费时间  %fs' % ( loss_vectorized, toc – tic )
grad_difference = np.linalg.norm( grad_naive - grad_vectorized, ord = 'fro' )
print '损失误差: %f' % np.abs( loss_naive - loss_vectorized )
print '梯度误差: %f' % grad_difference
```

其检验结果大致应该如下所示。

softmax_loss_vectorized 代码检验运行结果：

```
显式循环版 loss: 2.330326e+00      花费时间  0.045000s
向量化版本 loss: 2.330326e+00      花费时间  0.004000s
损失误差: 0.000000
梯度误差: 0.000000
```

2.6.5　最小批量梯度下降算法训练 Softmax 分类器

完成了 Softmax 的核心代码后，接下来我们就使用最小批量梯度下降算法训练 Softmax 分类器。在训练阶段，该过程十分简单，基本上就是采样数据，然后调用 Softmax 函数计算梯度，之后再更新权重，然后重复上述执行过程。如下所示，为该过程的算法伪代码。

Softmax 训练过程伪代码：

```
输入：数据、数据类标、学习率、权重衰减因子、迭代次数和批量大小。
For i in 迭代次数：
    {
根据批量大小进行数据采样；
计算当前采样数据的损失值、梯度值；
存储当前损失值；
更新权重；
}
返回：历史损失值
```

需要注意的是，我们的计数是从 0 开始的，10 分类任务其 y 的最大值为 9，因此 num_classes=np.max(y)+1。在采样时，重复采样或非重复采样都可以接受，但重复采样的执行效率要高些，可以使用重复采样加快执行效率，并且其对于梯度影响可以忽略不计。接下来就打开“DLAction/classifiers/chapter2/softmax.py”文件，对 train()函数进行代码填充工作。

最小批量梯度下降算法训练代码块：

```
def train( self, X, y, learning_rate = 1e-3, reg = 1e-5, num_iters = 100,
           batch_size = 200, verbose = False ):
    num_train, dim = X.shape
    # 我们的计数是从 0 开始，因此 10 分类任务其 y 的最大值为 9。
    num_classes = np.max( y ) + 1
    if self.W is None:
        # 初始化 W。
        self.W = 0.001 * np.random.randn( dim, num_classes )
    # 存储每一轮的损失结果 W。
    loss_history = [ ]
    for it in xrange( num_iters ):
        X_batch = None
        y_batch = None
        ##############################################################
        #                          任务：                             #
        #     从训练数据 X 中采样大小为 batch_size 的数据及其类标，        #
        #     并将采样数据及其类标分别存储在 X_batch，y_batch 中。        #
        #        X_batch 的形状为 ( dim,batch_size )，                  #
        #        y_batch 的形状为 ( batch_size )，                      #
        #     提示：可以使用 np.random.choice 函数生成 indices。         #
        #              重复采样要比非重复采样快许多 。                   #
        ##############################################################



        ##############################################################
        #                        结束编码                              #
        ##############################################################
        # 计算损失及梯度。
        loss, grad = self.loss( X_batch,  y_batch,  reg )
        loss_history.append( loss )
        # 更新参数。
        #########################################################
        #                      任务：                             #
        #              使用梯度及学习率更新权重。                    #
```

```
        ############################################################

        ############################################################
        #                          结束编码                           #
        ############################################################
        if verbose and it % 500 == 0:
            print '迭代次数 %d / %d: loss %f' % ( it, num_iters, loss )
    return loss_history
```

如果编码顺利，那损失函数应该会随着迭代次数的增加而减少。运行以下代码块，检验实现是否正确。

测试 Softmax 训练代码块：

```
from classifiers.chapter2.softmax import *
softmax = Softmax( )
tic = time.time( )
loss_hist = softmax.train( X_sample, y_sample, learning_rate = 1e-7,
                           reg = 5e4, num_iters = 3500, verbose = True )
toc = time.time( )
print '花费时间 %fs' % ( toc – tic )
```

其结果如下所示。

Softmax 训练结果：

```
迭代次数 0 / 3500: loss 780.062853
迭代次数 500 / 3500: loss 7.035170
迭代次数 1000 / 3500: loss 1.993476
迭代次数 1500 / 3500: loss 1.920132
迭代次数 2000 / 3500: loss 1.916136
迭代次数 2500 / 3500: loss 1.906755
迭代次数 3000 / 3500: loss 1.918884
花费时间 15.643000s
```

为了更直观地说明，我们将损失函数可视化并观察其变化情况，代码如下所示，损失函数如图 2-8 所示。

Softmax 损失函数可视化代码块：

```
plt.plot( loss_hist )
plt.xlabel( 'Iteration number' )
plt.ylabel( 'Loss value' )
plt.show( )
```

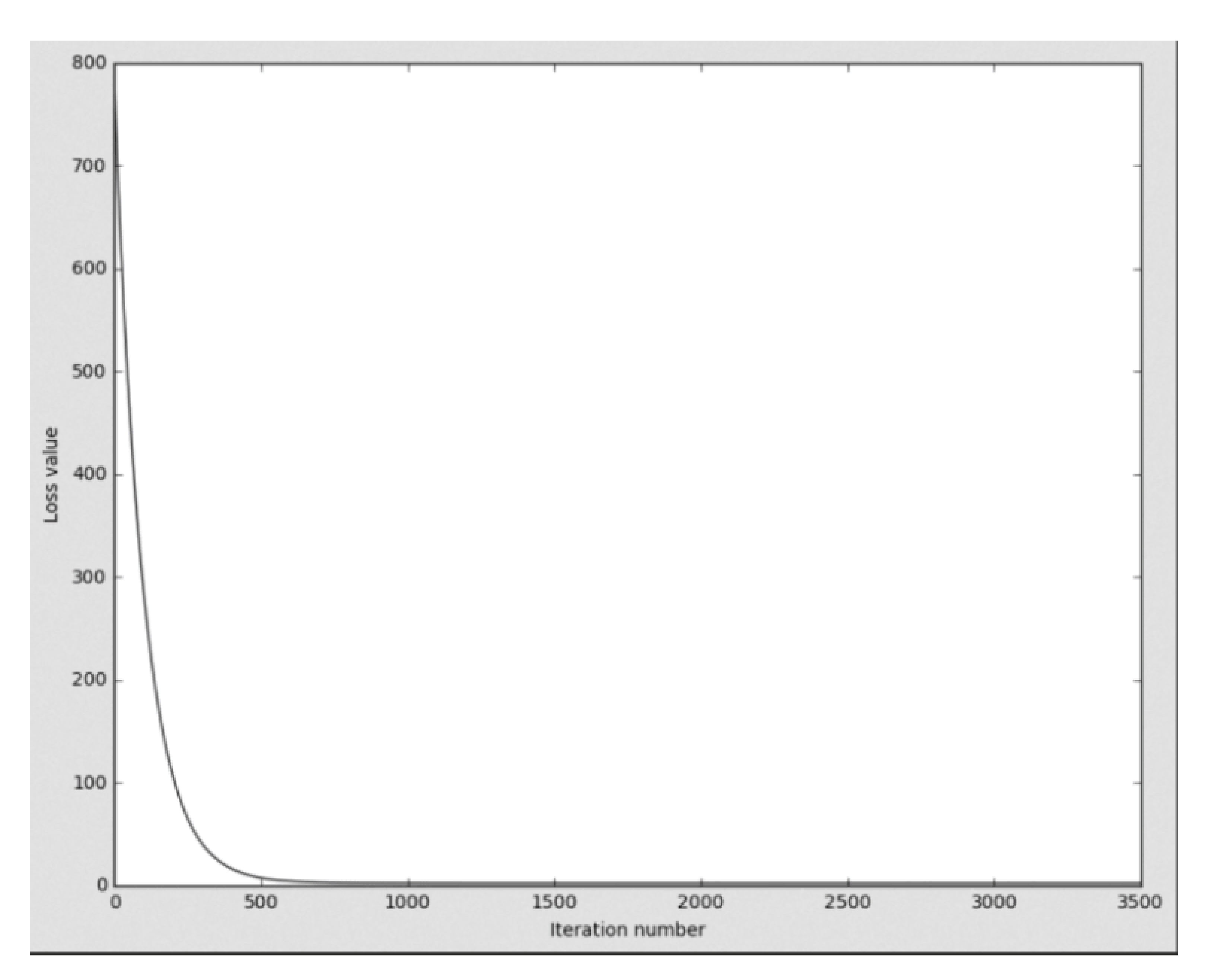


图 2-8 Softmax 损失函数变化情况

接下来我们编写 Softmax 的预测代码块。在预测阶段，我们不需要进行归一化概率，仅仅输出最高分所对应的类标号即可，该过程应该会很简单。一行代码就可以完成。

Softmax 分类器预测代码块：

```
def predict( self, X ) :
    """
    使用已训练好的权重预测数据类标。
    Inputs:
    - X:数据形状（N,D），表示 N 条数据，每条数据有 D 维。
    Returns:
    - y_pred：形状为( N, )，数据 X 的预测类标，y_pred 是一个长度为 N 的一维数组，
    每一个元素是预测的类标整数。
    """
    y_pred = np.zeros( X.shape[ 0 ] )
    ################################################################
    #                        任务：                                 #
    #               执行预测类标任务，将结果存储在 y_pred。           #
    #################################################################



    #################################################################
    #                        结束编码                               #
```

```
    ################################################################
    return y_pred
```

完成上述代码后，就可以测试效果，我们输出训练数据量、验证数据量、训练正确率和验证正确率。其代码块及输出结果如下所示。

```
测试训练集，验证集的精度代码块：
y_train_pred = softmax.predict( X_sample )
print y_train_pred.shape
print '训练数据量: %f  训练正确率: %f' % ( X_sample.shape[ 0 ],
                          np.mean( y_sample == y_train_pred ), )
y_val_pred = softmax.predict( X_val )
print '验证数据量:%f  验证正确率: %f' % ( X_val.shape[ 0 ],
np.mean( y_val == y_val_pred ), )
(250L,)
训练数据量:250.000000     训练正确率: 0.568000
验证数据量:100.000000     验证正确率: 0.240000
```

通过结果我们发现，训练精度和验证精度都不高，并且还出现了过拟合现象，接下来我们就开始使用超参数进行调整，训练出一个最佳模型。

2.6.6 使用验证数据选择超参数

深度学习工程师，开个玩笑，也可以被称为“调参工程师”，我们有太多的超参数可以选择。就目前而言，**学习率、权重衰减惩罚因子、批量大小和迭代次数**都可以称为**超参数**。超参数对最终的训练结果影响显著，并且不同的超参数组合，其结果也千差万别。接下来我们将使用学习率以及权重衰减因子作为超参数进行选择，请让你的机器飞起来吧！

我们将固定批量大小以及迭代次数，其中 batch_size=50，num_iters=300。

对于学习率和惩罚因子，使用逐步缩小范围的方式，来挑选超参数。其学习率为：learning_rates=np.logspace(-9,0,num=10)。

```
In [2]: learning_rates=np.logspace(-9, 0, num=10)

In [3]: learning_rates
Out[3]:
array([  1.00000000e-09,   1.00000000e-08,   1.00000000e-07,
         1.00000000e-06,   1.00000000e-05,   1.00000000e-04,
         1.00000000e-03,   1.00000000e-02,   1.00000000e-01,
         1.00000000e+00])
```

惩罚因子为：regularization_strengths=np.logspace(0,5,num=10)。

```
In [4]: regularization_strengths=np.logspace(0, 5, num=10)

In [5]: regularization_strengths
Out[5]:
array([  1.00000000e+00,   3.59381366e+00,   1.29154967e+01,
         4.64158883e+01,   1.66810054e+02,   5.99484250e+02,
         2.15443469e+03,   7.74263683e+03,   2.78255940e+04,
         1.00000000e+05])
```

其完整的训练代码和训练结果如下所示。

使用权重衰减因子和学习率作为超参数训练 Softmax 分类器:

```
# 使用验证集调整超参数（权重衰减因子，学习率）。
from classifiers.chapter2.softmax import *
results = { }
best_val = -1
best_l = 0
best_r = 0
best_softmax = None
learning_rates = np.logspace( -9, 0, num = 10 )
regularization_strengths = np.logspace( 0, 5, num = 10 )
batch_size = [ 50 ]
num_iters    = [ 300 ]
for b in batch_size :
    for n in num_iters :
        for l in learning_rates :
            for r in regularization_strengths :
                softmax = Softmax( )
                loss_hist = softmax.train(X_sample, y_sample, learning_rate = l, reg = r,
num_iters = n, batch_size = b, verbose = False)
                y_train_pred = softmax.predict( X_sample )
                train_accuracy= np.mean( y_sample == y_train_pred )
                y_val_pred = softmax.predict( X_val )
                val_accuracy = np.mean( y_val == y_val_pred )
                results[ ( l, r ) ] = ( train_accuracy, val_accuracy )
                if ( best_val < val_accuracy ):
                   best_val = val_accuracy
                   best_softmax = softmax
                   best_l = l
                   best_r =r
for lr, reg in sorted( results ) :
    train_accuracy, val_accuracy = results[ ( lr, reg ) ]
    print 'lr %e reg %e 训练精度: %f  验证精度: %f' % (lr, reg, train_accuracy, val_accuracy)
```

```
print '最佳学习率为:%e 最佳权重衰减系数为:%e 其所对应的验证精度为: %f' % ( best_l, best_r, best_val )
```

```
训练结果：
........
lr 1.000000e+00 reg 1.291550e+01 train accuracy: 0.124000 val accuracy: 0.070000
lr 1.000000e+00 reg 4.641589e+01 train accuracy: 0.124000 val accuracy: 0.070000
.........
lr 1.000000e+00 reg 1.000000e+05 train accuracy: 0.124000 val accuracy: 0.070000
最佳学习率为:1.000000e-05 最佳权重衰减系数为:3.593814e+00 其所对应的验证精度为: 0.280000
```

为了更直观地展示训练结果，我们将训练精度以及验证精度进行可视化比较，使用颜色表示训练性能，颜色越红则效果越好，颜色越蓝则效果越差。以下为可视化结果的代码块，图 2-9 为可视化训练结果。

```
可视化 Softmax 分类器训练结果代码块：
import math
x_scatter = [ math.log10( x[ 0 ] ) for x in results ]
y_scatter = [ math.log10( x[ 1 ] ) for x in results ]
# 绘制训练数据精度。
marker_size = 100
colors = [ results[ x ][ 0 ] for x in results ]
plt.subplot( 2, 1, 1 )
plt.scatter( x_scatter, y_scatter, marker_size, c = colors )
plt.colorbar( )
plt.xlabel( 'log learning rate' )
plt.ylabel( 'log regularization strength' )
plt.title( 'CIFAR-10 training accuracy' )
# 绘制验证数据精度。
colors = [ results[ x ][ 1 ] for x in results ]
plt.subplot( 2, 1, 2 )
plt.scatter( x_scatter, y_scatter, marker_size, c = colors )
plt.colorbar( )
plt.xlabel( 'log learning rate' )
plt.ylabel( 'log regularization strength' )
plt.title( 'CIFAR-10 validation accuracy' )
plt.show( )
```

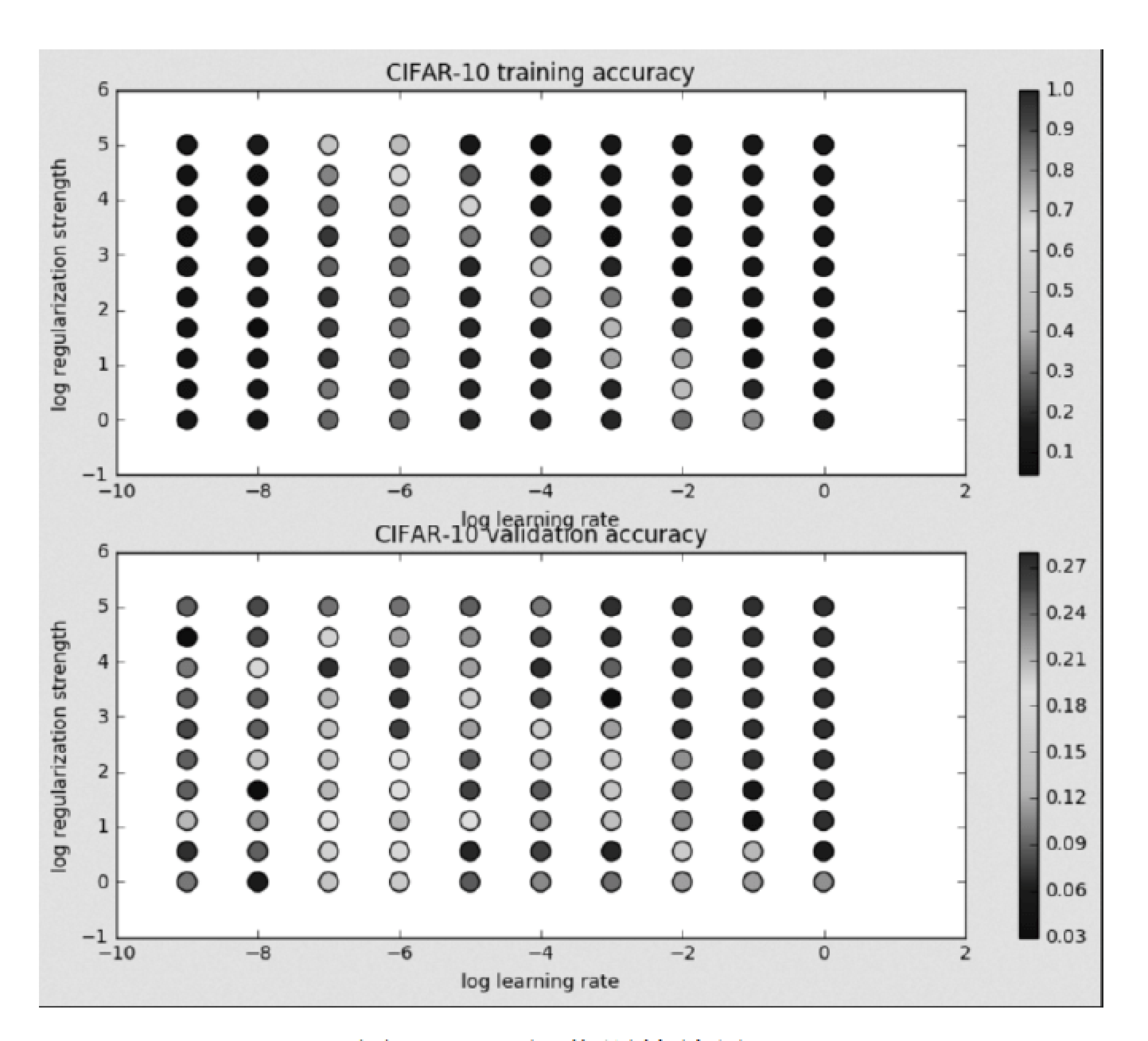


图 2-9 可视化训练结果一

从可视化结果可以清晰地看出，学习率大约在[1e-6,1e-4]之间效果显著，惩罚因子在[1e0,1e4]之间效果较好，并且惩罚因子越小可能效果越好。因此我们下一步就在这个范围内继续缩小。代码如下所示，图 2-10 是新的训练结果。

```
learning_rates = np.logspace( -6, -4, num = 10 )
regularization_strengths = np.logspace( -1, 4, num = 5 )
```

```
最佳学习率为:7.742637e-06 最佳权重衰减系数为:1.000000e-01 其所对应的验证精度为:0.310000
```

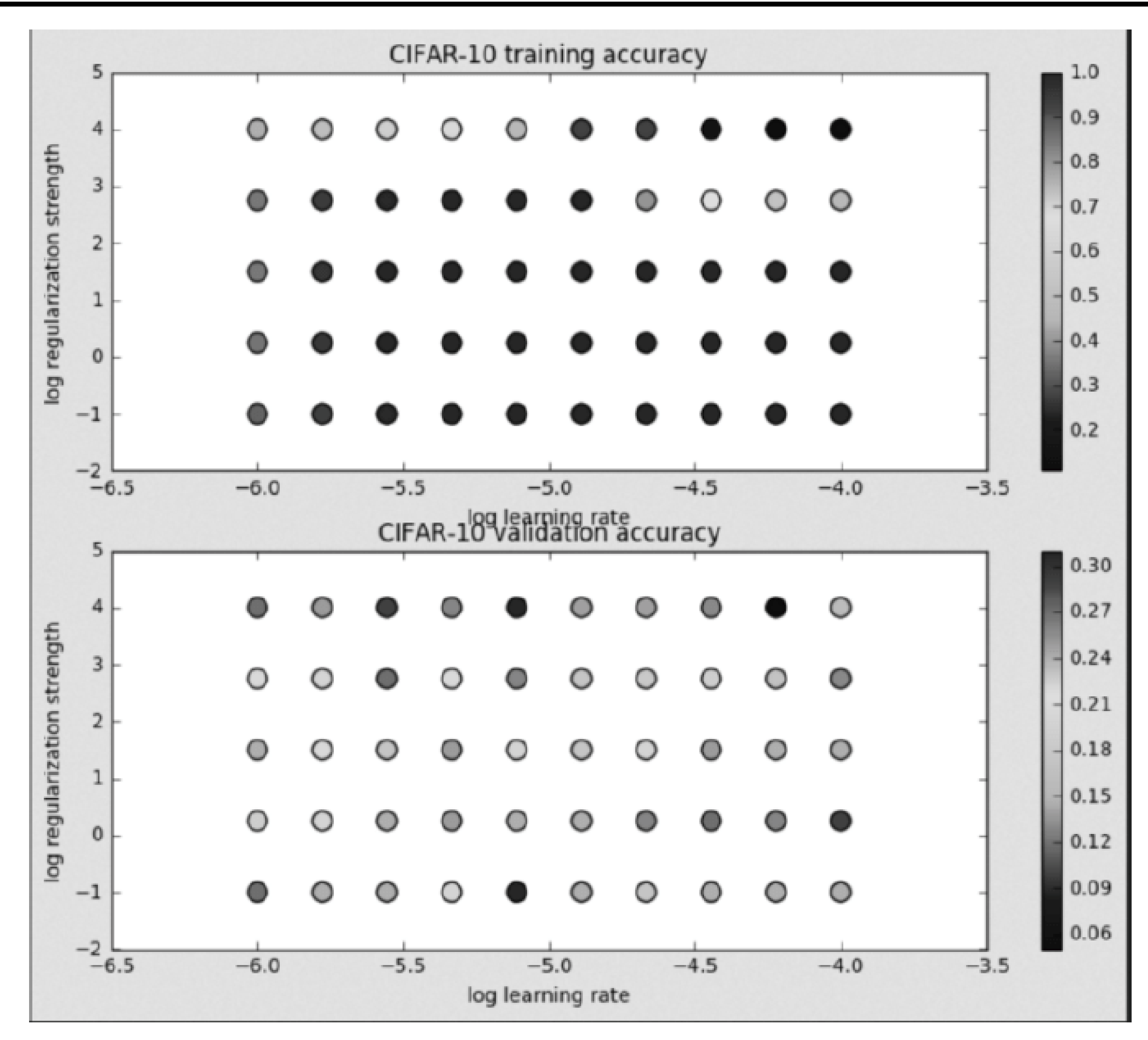


图 2-10 可视化训练结果二

我们再次设置选择范围，如下所示，新的训练结果如图 2-11 所示。

```
learning_rates = np.logspace( -5, -4, num = 10 )
regularization_strengths = np.logspace( -2, 3, num = 5 )
```

```
最佳学习率为:1.668101e-05  最佳权重衰减系数为:3.162278e+00  其所对应的验证精度为:0.310000
```

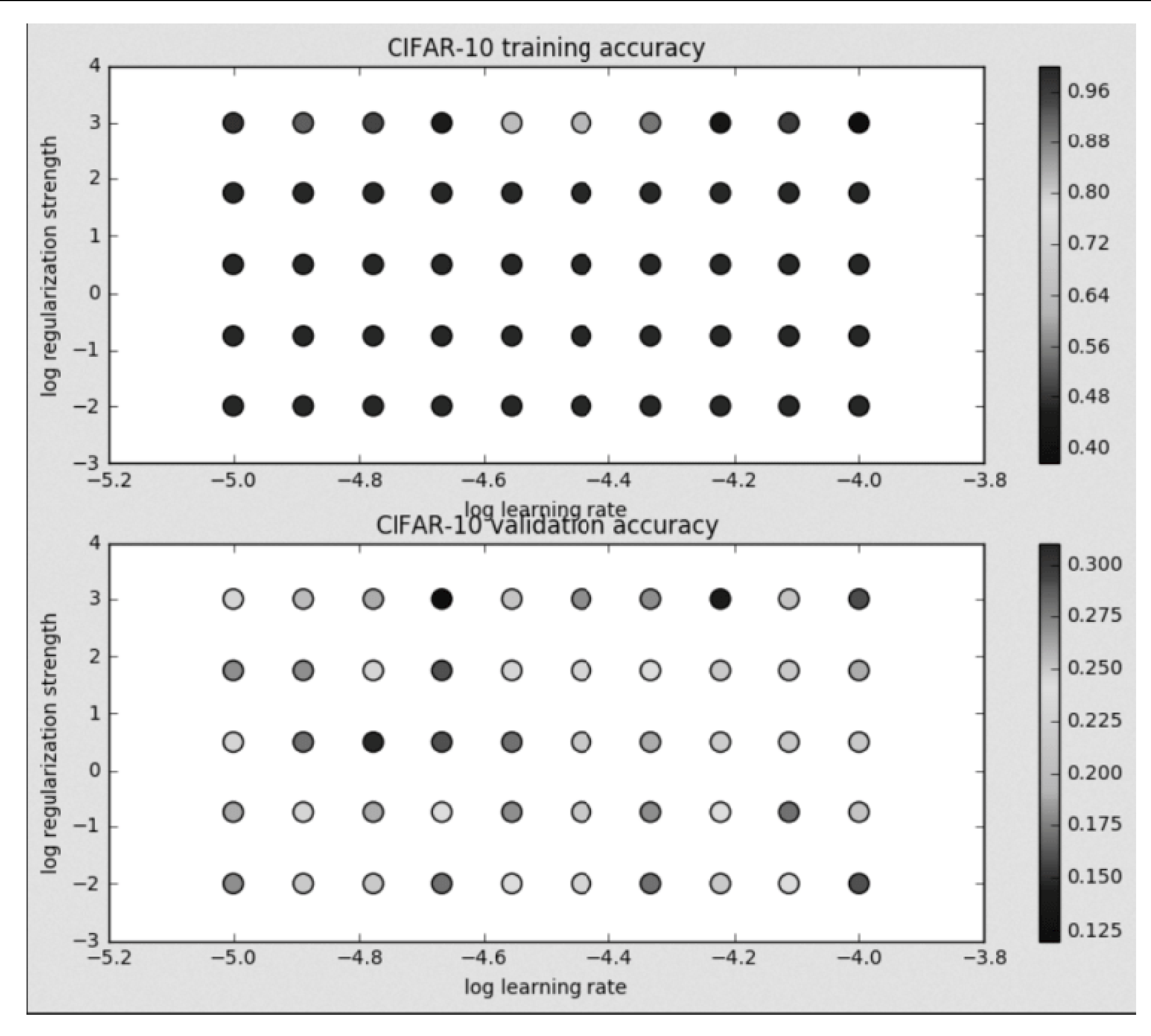


图 2-11　可视化训练结果三

此时我们的训练精度几乎都接近了 100%，但最佳的验证精度也只有 0.31。那最终的测试结果如何呢？如下所示是我们的 Softmax 测试结果代码块，非常令人沮丧，测试集精度只有 0.215。

softmax 测试结果代码块：

```
# 在测试数据集上评估最佳 Softmax 分类器。
y_test_pred = best_softmax.predict( X_test )
test_accuracy = np.mean( y_test == y_test_pred )
print '测试集精度: %f' % test_accuracy
```

```
测试集精度: 0.215000
```

实在不好意思，这是恶趣味的作者故意要的一个小花招，那你知道问题出在哪吗？如果你足够认真和细心的话，可能早就发现了，那就是我们训练的数据量实在太小了。上述的测试中，我们仅仅使用了 250 条数据进行训练，不管我们如何优化，过拟合情况都很难避免，之所以这么做，是想让你注意一个微小而又重要的知识，那就是数据量的问题。

当数据较少时，过拟合风险就越高，即使非常小心地选择超参数，其效果也不会好，这是一个令人伤感的消息。但其实也有一个好消息，那就是当数据量足够大时，过拟合现象就

很难发生，这就使得训练数据、验证数据与测试数据之间的差值会越来越接近。在某些数据量特别巨大的情况下，可能只需关注如何将训练数据错误率降到足够低即可。超参数的选择是一个非常耗时、耗力的过程，因此我们可以先使用较小的数据去粗略地选择超参数的取值范围，这样可以节约训练时间。但在较小数据中表现最好的超参数，不一定在数据量较大时同样表现得更好，因此需要注意数据量的把控，但这是一个非常依赖于经验的问题，需要从大量的实验中自己积累经验，进行不断地尝试。记住，千万不要怕错。

现在我们将重新载入数据集，使用完整的 49000 条数据进行训练，1000 条数据作为验证。现在舞台交给你了，请尽一切可能训练出一个最好的测试结果，你可以尽可能地调整现在所提供的 4 种超参数，如果计算能力和自己的耐心允许，也可以尝试下交叉验证。如果你足够努力，你的测试正确率可以超过 0.35。重新载入数据代码块和训练代码块分别如下所示。

重新载入数据代码块：

```
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data( num_training = 49000, num_validation =
1000, num_test = 10000 )
```

```
Train data shape:   (49000L, 3073L)
Train labels shape:   (49000L,)
Validation data shape:   (1000L, 3073L)
Validation labels shape:   (1000L,)
Test data shape:   (10000L, 3073L)
Test labels shape:   (10000L,)
```

Softmax 训练代码块：

```
from classifiers.chapter2.softmax   import Softmax
results = { }
best_val = -1
best_softmax = None
#################################################################
#                              任务：                              #
#                 使用全部训练数据训练一个最佳 softmax。               #
#################################################################
learning_rates = [ ]
regularization_strengths = [ ]
learning_rates = [ ]
regularization_strengths = [ ]




#################################################################
#                              结束编码                            #
#################################################################
```

```
for lr, reg in sorted( results ):
    train_accuracy, val_accuracy = results[ ( lr, reg ) ]
    print 'lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy )
print '最佳验证精度为: %f ' % best_val
```

另一种更直观检验模型好坏的方法是可视化模型参数，比如我们识别图片“马”，那模型的参数其实就是图片的“马模板”，也可以通过可视化参数去反推数据情况。比如我们的图片中如果存在大量“马头向左”与“马头向右”的图片，那训练出来的模型参数很可能就变成了“双头马”。同理，如果图片中汽车的颜色大多数都是红色，那训练出的模型就可能是一辆“红车模板”。可视化参数代码块如下所示，可视化训练参数示意图如图 2-12 所示。

可视化参数代码块：

```
# 可视化学习到的参数：
w = best_softmax.W[ : -1, : ] # 移除偏置项。
w = w.reshape( 32, 32, 3, 10 )
w_min, w_max = np.min( w ), np.max( w )
classes = [ 'plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck' ]
for i in xrange( 10 ):
  plt.subplot( 2, 5, i + 1 )
  # 将权重缩放回 0~255。
  wimg = 255.0 * ( w[ : , : , : , I ].squeeze( ) - w_min ) / ( w_max - w_min )
  plt.imshow( wimg.astype( 'uint8' ) )
  plt.axis( 'off' )
  plt.title( classes[ i ] )
```

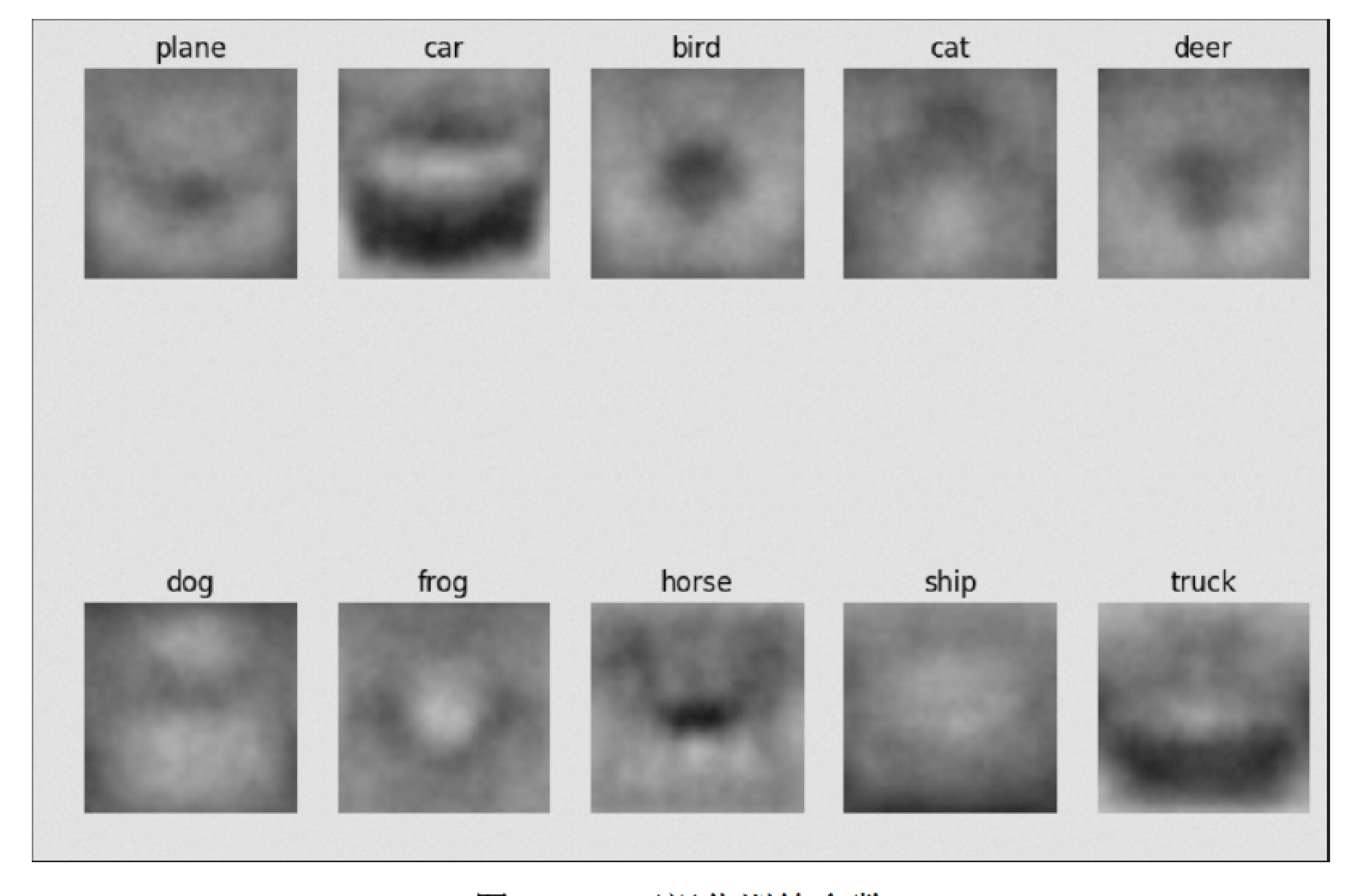


图 2-12　可视化训练参数

2.7 参考代码

softmax_loss_naive 代码块：

```
def softmax_loss_naive( W, X, y, reg ) :
    loss = 0.0
    dW = np.zeros_like( W )
    #################################################################
    # 任务：使用显式循环实现 Softmax 损失值 loss 及相应的梯度 dW。        #
    # 提示：如果不慎，将很容易造成数值上溢。别忘了正则化。                #
    #################################################################
    num_train = X.shape[ 0 ]
    num_class = W.shape[ 1 ]
    for i in xrange( num_train ) :
        s = X[ i ].dot( W )
        scores = s - max( s )
        scores_E = np.exp( scores )
        Z = np.sum( scores_E )
        scores_target = scores_E[ y[ i ] ]
        loss + = -np.log( scores_target / Z )
        for j in xrange( num_class ) :
            if j == y[ i ] :
                dW[ : , j ] + = -( 1 - scores_E[ j ] / Z ) * X[ i ]
            else :
                dW[ : , j ] + = X[ i ] * scores_E[ j ] / Z
    loss = loss / num_train + 0.5 * reg * np.sum( W * W )
    dW = dW / num_train + reg * W
    ###############################################################
    #                          结束编码                            #
    ###############################################################
    return loss, dW
```

softmax_loss_vectorized 代码块：

```
def softmax_loss_vectorized( W, X, y, reg ) :
    loss = 0.0
    dW = np.zeros_like( W )
    ###############################################################
    # 任务：不使用显式循环计算 Softmax 的损失值 loss 及其梯度 dW。    #
    # 提示：如果不慎，将很容易造成数值上溢。别忘了正则化。              #
    ###############################################################
```

```
    num_train = X.shape[ 0 ]
    s = np.dot( X, W )
    scores = s - np.max( s, axis =1, keepdims = True )
    scores_E = np.exp( scores )
    Z = np.sum( scores_E, axis = 1, keepdims = True )
    prob = scores_E / Z
    y_trueClass = np.zeros_like( prob )
    y_trueClass[ range( num_train ), y ] = 1.0      # （N,C)
    loss + = -np.sum( y_trueClass * np.log(prob)) / num_train + 0.5 * reg * np.sum(W*W)
    dW + = -np.dot( X.T, y_trueClass – prob ) / num_train + reg * W
    ##################################################################
    #                          结束编码                              #
    ##################################################################
    return loss, dW
```

最小批量梯度下降算法训练代码块：

```
    def train( self, X, y, learning_rate = 1e-3, reg = 1e-5, num_iters = 100,
            batch_size = 200, verbose = False ) :
        num_train, dim = X.shape
        num_classes = np.max( y ) + 1
        if self.W is None:
            self.W = 0.001 * np.random.randn( dim, num_classes )
        loss_history = [ ]
        for it in xrange( num_iters ) :
            X_batch = None
            y_batch = None
            ##########################################################
            #                       任务：                           #
            #     从训练数据 X 中采样大小为 batch_size 的数据及其类标   #
            #     并将采样数据及其类标分别存储在 X_batch，y_batch 中    #
            #         X_batch 的形状为( dim,batch_size )，             #
            #         y_batch 的形状为( batch_size )。                 #
            #     提示：可以使用 np.random.choice 函数生成 indices，    #
            #              重复采样要比非重复采样快许多。               #
            ##########################################################
            indices = np.random.choice( num_train, batch_size, False )
            X_batch = X[ indices, : ]
            y_batch = y [indices ]
            ##########################################################
```

```
            #                        结束编码                         #
            ############################################################
            loss, grad = self.loss( X_batch, y_batch, reg )
            loss_history.append( loss )
            ############################################################
            #                         任务:                           #
            #                   使用梯度及学习率更新权重                  #
            ############################################################
            self.W = self.W - learning_rate * grad
            ############################################################
            #                        结束编码                          #
            ############################################################
            if verbose and it % 500 == 0:
                print '迭代次数  %d / %d: loss %f' % ( it, num_iters, loss )
        return loss_history
```

Softmax 分类器预测代码块:

```
    def predict( self, X ) :
        y_pred = np.zeros( X.shape[ 0 ] )
        ############################################################
        #                         任务:                            #
        #             执行预测类标任务，将结果存储在 y_pred。           #
        ############################################################
        y_pred = np.argmax( X.dot( self.W ), axis = 1 )
        ############################################################
        #                        结束编码                           #
        ############################################################
        return y_pred
```

2.8　参考文献

[1] 周志华. (2016). 机器学习 : = Machine learning: 清华大学出版社.

[2] Mitchell, T. M., Carbonell, J. G., & Michalski, R. S. (2003). Machine Learning: McGraw-Hill.

[3] Bishop, C. M. (2006). Pattern Recognition and Machine Learning (Information Science and Statistics): Springer-Verlag New York, Inc.

[4] Murphy, K. P. (2012). Machine Learning: A Probabilistic Perspective: MIT Press.

[5] Wang, Z., & Bovik, A. C. (2009). Mean squared error: Love it or leave it? A new look at Signal Fidelity Measures. IEEE Signal Processing Magazine, 26(1), 98-117.

[6] Akaike, H. (1992). Information Theory and an Extension of the Maximum Likelihood Principle. Inter.symp.on Information Theory, 1, 610-624.

[7] Shore, J. E., & Johnson, R. W. (1980). Axiomatic derivation of the principle of maximum entropy and the principle of minimum cross-entropy. Information Theory IEEE Transactions on, 26(1), 26-37.

[8] Nakama, T. (2009). Theoretical analysis of batch and on-line training for gradient descent learning in neural networks. Neurocomputing, 73(1–3), 151-159.

[9] Li, M., Zhang, T., Chen, Y., & Smola, A. J. (2014). Efficient mini-batch training for stochastic optimization.

[10] Bottou, L. (2010). Large-Scale Machine Learning with Stochastic Gradient Descent: Physica-Verlag HD.

[11] Mairal, J., Bach, F., Ponce, J., & Sapiro, G. (2009). Online Learning for Matrix Factorization and Sparse Coding. Journal of Machine Learning Research, 11(1), 19-60.

[12] Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A., Jaitly, N., . . . Sainath, T. N. (2012). Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. IEEE Signal Processing Magazine, 29(6), 82-97.

[13] Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation, 1(1), 67-82.

[14] Krizhevsky, A. (2012). Convolutional Deep Belief Networks on CIFAR-10.

[15] Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. Paper presented at the International Joint Conference on Artificial Intelligence.

[16] Dietterich, T. G., & Bakiri, G. (1994). Solving multiclass learning problems via error-correcting output codes: AI Access Foundation.

第 3 章

前馈神经网络

故事从一个十分发达的地外文明开始，“球球星”住着科技高度发达的“球球人”。他们热爱和平，但也惧怕战争，于是就监听其他星系的文明，而地球就属于士兵小飞的监控范围。一天少校问士兵小飞，地球怎么样了。小飞就说，“地球文明还很原始，地球上最高级的物种，是一种名叫汽车的生物，汽车的速度很快，但其不靠阳光，也不会主动觅食，汽车专门奴役一种称为人的生物，当该生物进入汽车时，汽车就会获得燃料，然后就可以四处活动。当燃料用尽时，汽车就会吐出这种生物，让其四处补充能量，以便下一次循环利用。该生物还比较低级，经常还会因为自身速度过快，或因误食变质人类而失控，导致互相残杀，不知少校如何处置？”少校思考稍许之后说道，“他们内耗严重，过于愚蠢，暂不处理，继续暗中观察。”你可能会嘲笑士兵小飞的无知，明明是人在使用汽车，结果却观察到汽车奴役人类。但如果把“汽车和人”换成“人和基因”，那会不会让你后背发凉呢？

某种程度上，我们观察到的世界或许和“球球星”人观察到的地球差不多，我们即使找到了非常“合理”的解释，但其也不过是我们的幻象罢了。虽然很难想象一堆粉红色的肉团如何能造出缤纷的世界，但以现代科学的观点，人的智能行为，或者人们所说的“灵魂”，可能仅仅是由一些脑神经组合起来的行为而已。也许就如人工智能之父，马文·明斯基的那句名言一样，“大脑无非是肉做的机器而已（The brain happens to be a meat machine）”。

要解开大脑的秘密，可能还需要很多代人的不懈努力。如果说神经科学是以生物的方式去破解智能密码，那么深度学习就是以计算机的方式去寻找智能钥匙，并且深度学习与神经科学也是相辅相成，很多研究者经常在这两种学科间互换。但神经网络的研究已经不仅仅是

“大脑模型”而已，同时也遵循许多数学与工程原则，神经网络也被看成是一些并行的处理单元，因此“神经元（neuron）”有时也用“单元（unit）”替换，但这仅仅是文字游戏，希望不要被其弄晕。

在本章的 3.1 节中，我们将介绍**神经元**，其是神经网络的基本组成单元，就像人与人聚在一起组成了社会一样，研究社会的起点及终点都应该是人，而研究神经网络的起点也是从简单的神经元开始，但深度学习中的神经元并不是一个生物概念，和生物神经元的差别也非常大。

在本章的 3.2 节中，我们将介绍**前馈神经网络**，该网络是通用的深度学习模型，并且后续章节的卷积神经网络，循环神经网络都是该网络的变种，因此该网络是更高级神经网络的基础，你需要认真学习。

本章的 3.3 节中，我们会详细地介绍**反向传播算法（BP 算法）**，该算法自 20 世纪 80 年代起就一直统治着深度学习，其算法思想其实非常简单，但对于初学者而言可能有些摸不着头脑，因此在本小节中会详解该算法的每一步，让你充分理解该算法。

而在 3.4 节的编程实践环节中，我们将逐步地从简单的仿射传播，到单层传播，再到浅层网络，最后到全连接神经网络，一步一步构造出完整的深层神经网络。你需要耐心且独立地完成每一小节的任务，同样在本章最后，我们也会给出参考代码来帮助你学习。

3.1 神经元

神经细胞利用“电-化学”过程进行信号交换，但信号在大脑中实际怎样传输是一个相当复杂的过程。深度学习并不模拟完整的神经元，为了方便计算机使用，我们简化其模型，通常只借用神经科学中的一些概念去帮助我们理解。

我们将大脑的神经细胞简化为两种状态：兴奋（fire）和不兴奋（即抑制），发射信号的强度不变，变化的仅仅是频率。神经细胞利用我们还不知道的方法，把所有输入进神经元的信号进行累加，如果全部信号的总和超过某个阈值，就会刺激神经细胞进入兴奋状态，这时就会有电信号发送给其他神经细胞。如果信号总和没有达到某个阈值，神经细胞就会处于抑制状态，不向周围发送信号。虽然这样的解释有点过于简单，但已能满足我们的目的了。

如图 3-1 所示，为神经元模型。该模型主要由两部分构成，第一部分进行信号累加，就如我们第 2 章介绍过的**得分函数**（score function），该部分仅对输入信号（输入数据）进行累加求和，如式(3.1)所示，每一权重 w_i 对应着相应的数据维度 x_i，而 w_0 作为我们的**偏置项**（bias）就相当于函数的截距项或常数项。为了简化描述，我们通常增加第 0 维输入，该输入为常数 1，将偏置项作为第 0 维的权重处理，在几何学中，我们也将这种计算称为**仿射变换**（Affine Transformation）。

$$Z = \text{bias} + \sum_{i=1}^{m} x_i w_i = \sum_{i=0}^{m} x_i w_i \tag{3.1}$$

第二部分为**激活函数**（Activation Function），这是神经元的关键，通常使用某类激活函数，就被称为某类神经元。如果我们的激活函数使用 Sigmoid 激活函数，那该神经元就相当于 Logistic 回归[1]。

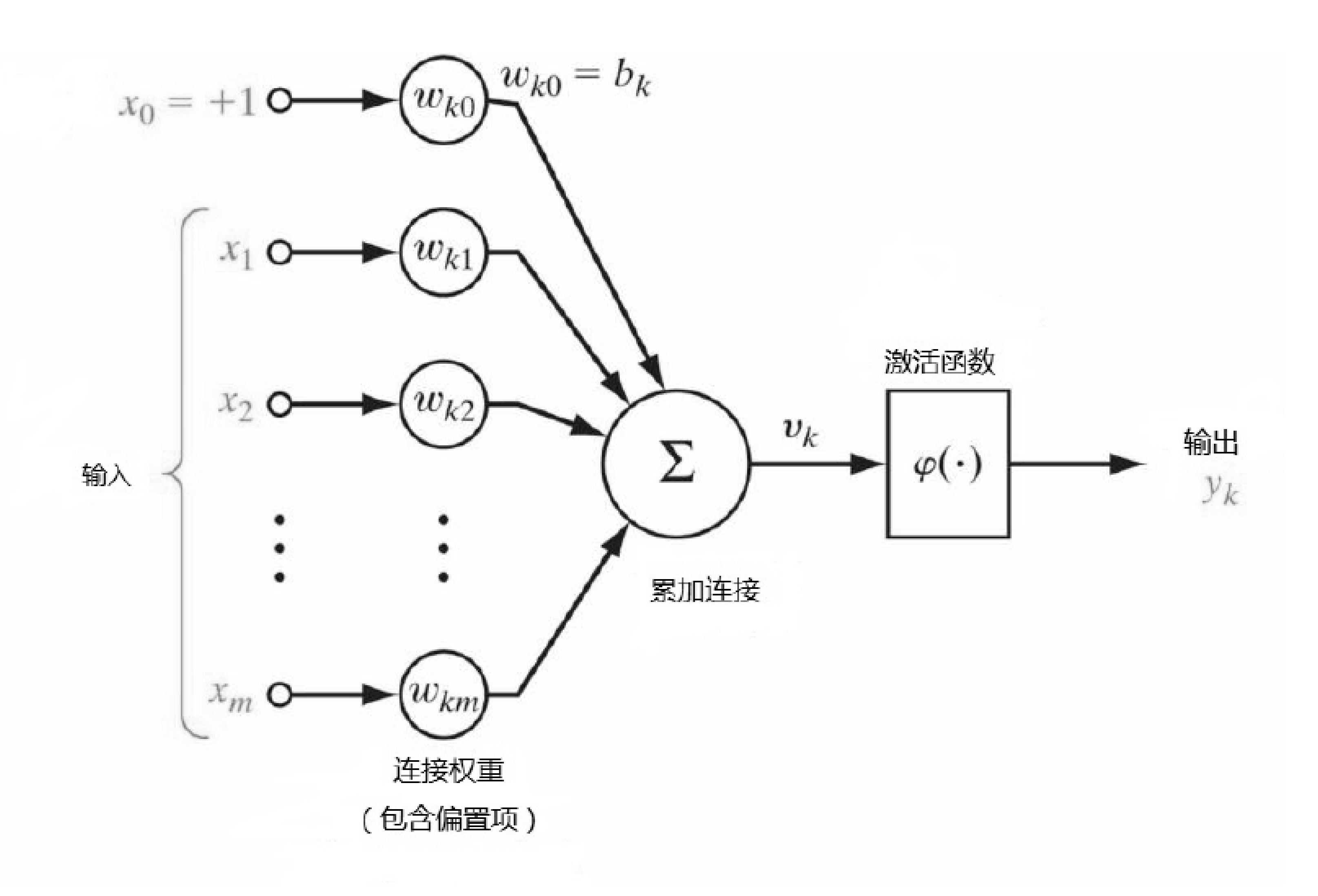


图 3-1　神经元模型

如图 3-2 所示，最开始的激活函数使用的是**阈值**（Threshold）[2]激活函数，就如我们对神经元的最简化描述一样，如果阈值超过了 0，那函数就恒定输出 1；如果小于 0，函数就恒定输出 0。该函数虽然不能求导，但使用**感知机收敛理论**[3]（可以说是随机梯度下降的鼻祖），依然可以学习**线性可分**的数据。但该神经元无法在多层感知机中学习基于梯度的优化算法，因此很少应用在现代神经网络中。

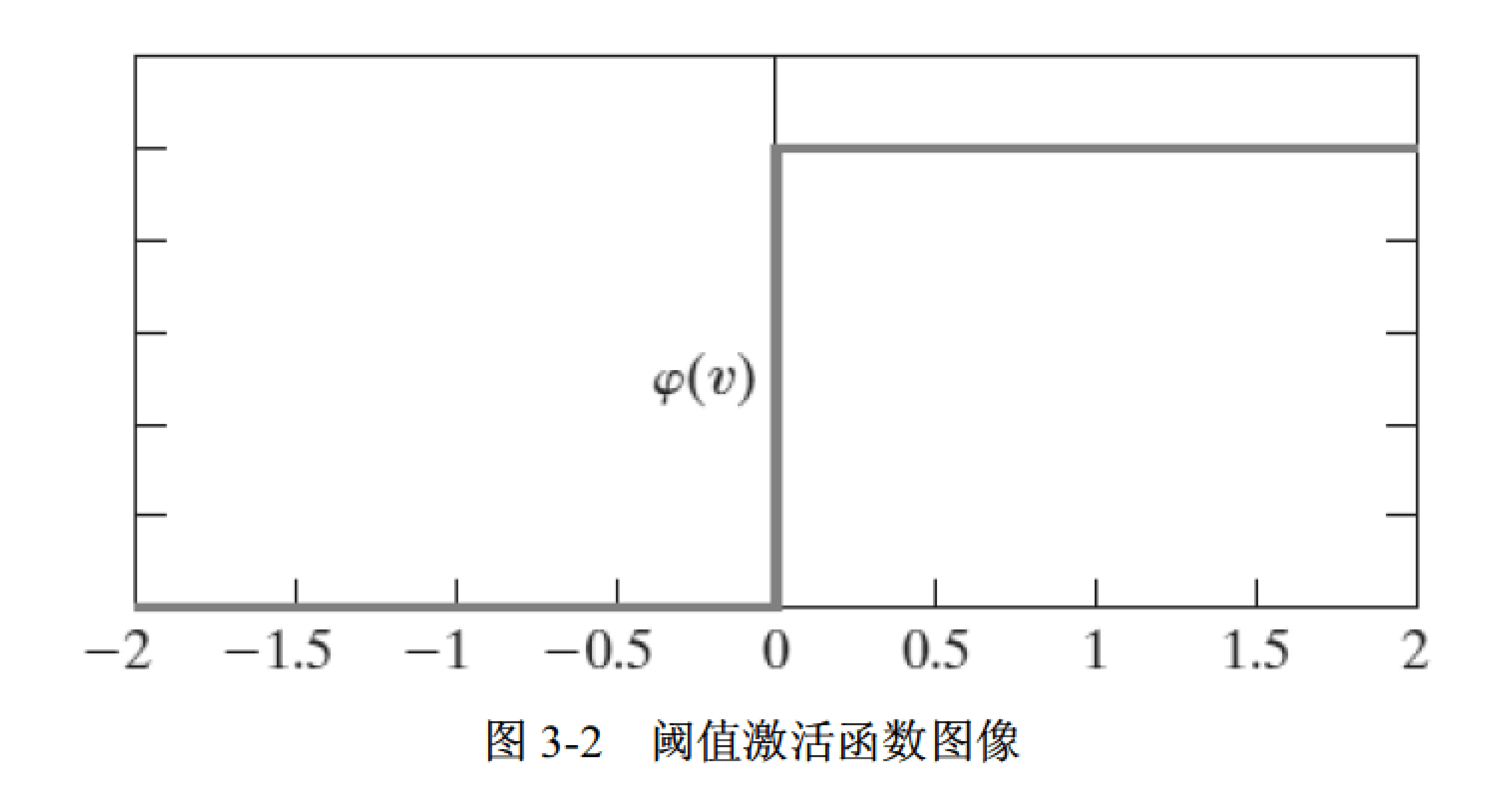


图 3-2　阈值激活函数图像

3.1.1　Sigmoid 神经元

以上的阈值函数已经可以很好地模拟神经元了，但其最致命的缺点（无法求导）也使得它无法应用到神经网络中。为了克服这一障碍，就引进了一种“软阈值”函数，也就是如图 3-3 所示的 Sigmoid 函数[4]。式（3.2）为该函数的表达式，如式（3.3）所示，Sigmoid 函数还有着非常简洁的导数表达式。

$$f(x) = (1 + \mathrm{e}^{-x})^{-1} \tag{3.2}$$

Sigmoid 函数的导函数。

$$f'(x) = f(x)(1 - f(x)) \tag{3.3}$$

在很长一段时间内，Sigmoid 神经元都是神经网络的默认配置，其不但能很好地模拟生物神经元的特点，并且还有着很和谐的函数性质，但它也有着一个过去人们比较忽视的缺点，那就是**易饱和性**（saturation），当输入值非常大或者非常小的时候，这些神经元的梯度就接近于 0。从图 3-3 中函数的变化趋势可以看出，当输入值远离 0 时，其函数会很快变得平稳，而这种平稳的代价就是梯度接近于 0。也就是说，当我们试图修改权重时，几乎得不到梯度进行学习。就如同一杯几乎饱和了的盐水，想要让其变淡，但每次只能加入几滴水。因此，使用 Sigmoid 神经元时，尤其需要注意参数的初始值来尽量避免饱和情况。如果初始值很大的话，大部分神经元可能都会处在饱和状态，这会导致网络变得很难学习，此时的学习就像“清水滴石”“铁杵磨针”，对于初学者而言或许没有太多感触，但当你进行实验时，就会发现这有多糟糕。

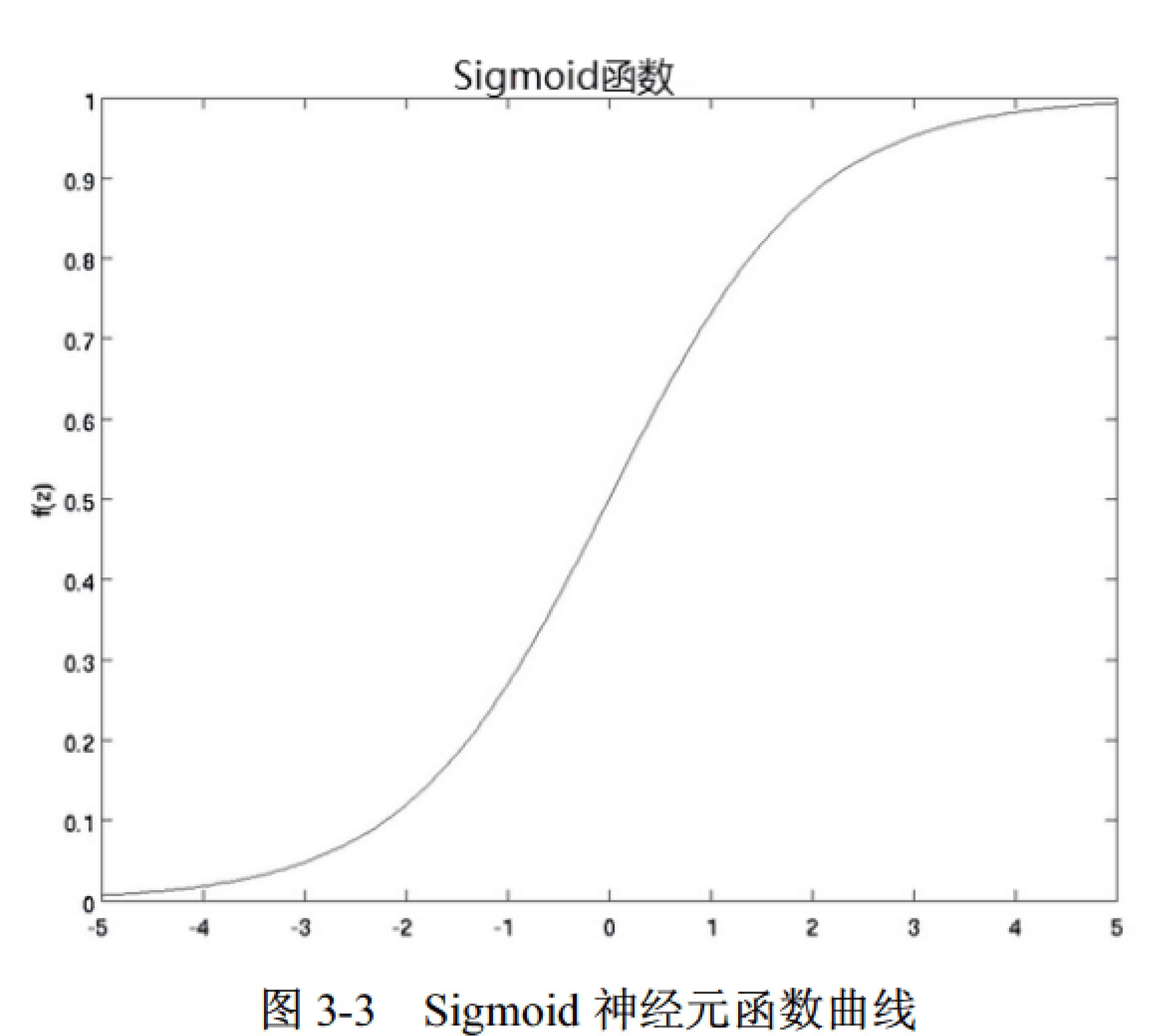


图 3-3　Sigmoid 神经元函数曲线

Sigmoid 神经元还有一个小缺点，那就是其输出的期望不是 0，这是不可取的，因为这会导致后一层的神经元将得到上一层输出的非 0 均值信号作为输入，这一知识点将会在第 5 章**批量归一化**（Batch Normalization）[5]章节中重点描述。

3.1.2　Tanh 神经元

如图 3-4 所示，为双曲线正切（Hyperbolic Tangent）[6]激活函数的函数图像。该函数的取值为(-1,1)，由于其函数期望为 0，因此可算作是 Sigmoid 神经元的一个小改进，通常在需要使用 Sigmoid 神经元的情况下，双曲线正切神经元是一个比较好的替代。但该神经元依然存在易饱和的性质，如式（3.4）所示，为该激活函数的表达式。

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.4}$$

虽然该函数看着比较复杂，但如式（3.5）所示，该函数化简之后其实就是 Sigmoid 函数

的一个变形。

$$\tanh(x) = 2\text{sigmoid}(2x) - 1 = 2(1 + e^{-2x})^{-1} - 1 \tag{3.5}$$

如式（3.6）所示，是该函数的导数，其导数为 1 减去该函数自身的平方。

$$\tanh'(x) = 1 - \tanh^2(x) \tag{3.6}$$

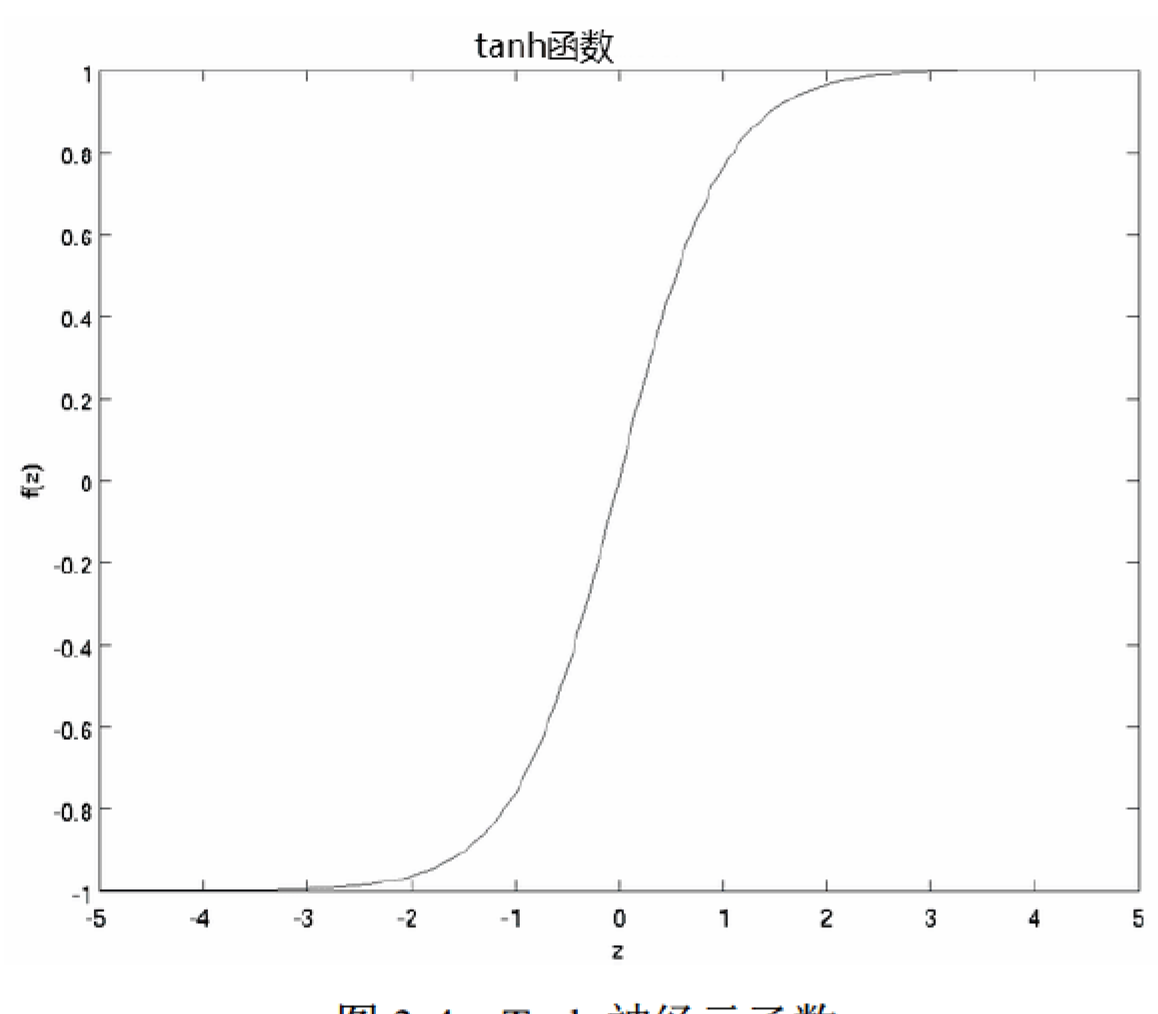


图 3-4　Tanh 神经元函数

3.1.3　ReLU 神经元

从 2006 年至今可以被称为神经网络的第三次高潮，大量的研究人员在此期间提出了重要的方法，但若非要列举几个最重要的突破，**修正线性单元**（Rectified Linear Units，ReLU）[7]的使用必然位列其中。在 2006 年之前，深度学习就像是机器学习领域的异类，过高的模型复杂度，使其难以进行数学分析，大量的训练“技巧”让外人难以进入该领域，非常容易过拟合又让人看不到希望。

当时深度学习最严重的问题就是**梯度消失问题**，模型的层数较多时，甚至只有三四层时，模型几乎就无法训练了。而以 Hinton 为首的深度学习研究者终于使用**逐层贪婪非监督预训练**学习的方法缓解了梯度消失这一问题。由于每一层都有较好的初始化，即使神经网络的底层依然无法得到有效的训练，但也能有一个非常好的训练结果。于是研究深度学习的“异类”们才渐渐地得到了鲜花和掌声，但逐层贪婪预训练方法并没有解决梯度消失问题，梯度消失始终就像是悬挂在深度学习头顶的“达摩克里斯之剑”。

故事并不像小说那样，主角都需要骨骼惊奇，万中无一，但故事又如人们期望的那样，救世主总是命中注定的。这就是本节将重点推介的平凡主角——**修正线性单元**（Rectified Linear Units，ReLU）。我喜欢戏称 ReLU 为“平凡的天选之人”，所谓平凡，指的是该神经元的简单；所谓的“天选之人”，指的是该神经元是更加接近生物神经元的模型。

深度学习早已脱离了“模拟大脑”的僵化描述，但深度学习的每一次重大突破，又和我们对自身大脑的认知脱不了关系。我们对于大脑、神经元的过简描述并不能有效地帮助我们。

同样地，我们对于神经元的过于复杂或过充分模拟也无法有效地帮助我们，如何折中并找到平衡，我们或许只能是“摸着石头过河”。就 ReLU 而言，比其简单的模型有过，比其复杂的模型也多不胜举，随着时间的推移，研究人员“众里寻他千百度”，提出了各种复杂“高深”的观点及算法，但蓦然回首，找到了如此简单又如此合理的 ReLU，如图 3-5 所示。

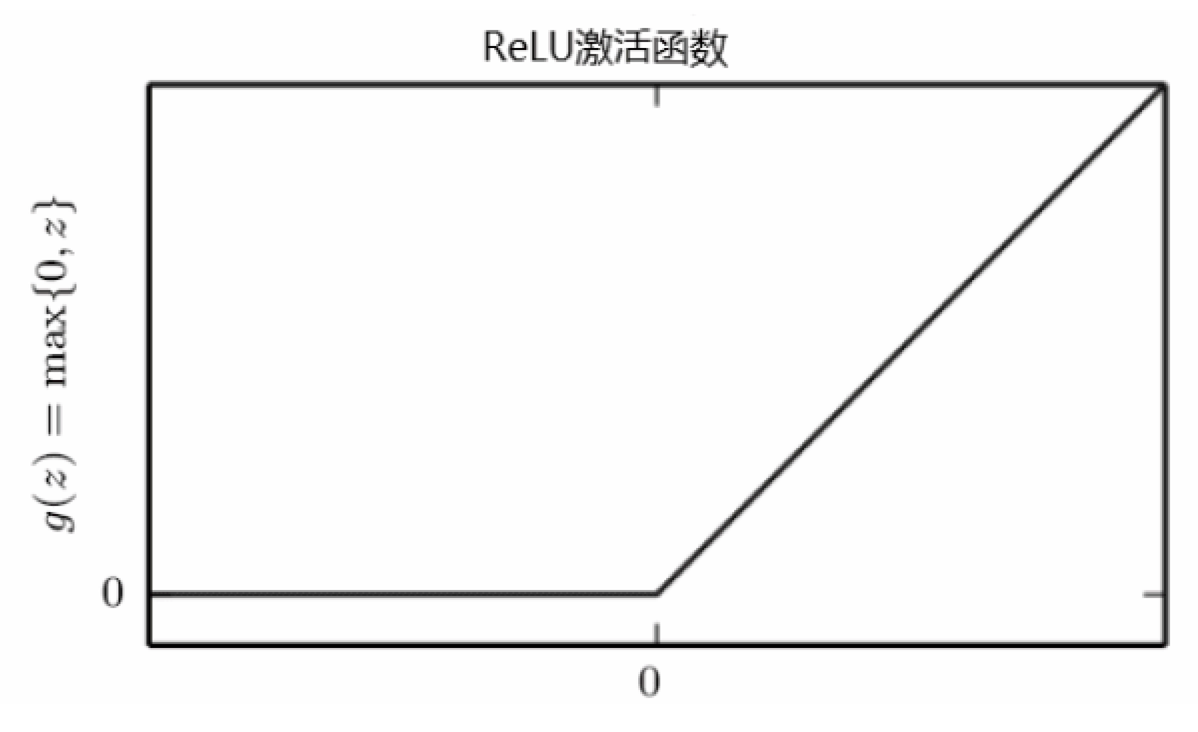


图 3-5　修正线性单元函数图像

看完图 3-5 所示的修正线性单元函数图像，不知你是什么感觉，“这不就是被阉割的线性函数吗？有什么了不起的。”如果你有这样的想法那也很正常，如式（3.7）所示，该函数正半部分仅是一次函数的正半部分，负半部分是 x 轴的负半部分。

$$f(x) = \begin{cases} x, & x \geqslant 0 \\ 0, & x < 0 \end{cases} \tag{3.7}$$

我们知道线性单元组合后的模型能力依然是线性能力，那首先需要质疑的就是，这被“砍了半截”的线性单元又有多强的能力呢？为了回答这个问题，我们使用致使神经网络第一次大衰败的“异或”问题作为案例。

如图 3-6 所示，有{ (1,1), (1,0), (0,1), (0,0) }4 个数据，数据(1,1)和数据(0,0)属于同一类，我们用“0”表示；而数据(1,0)和数据(0,1)属于同一类，我们用“1”表示。就相当于我们的数据有两维，若两维值相同就输出“0”，相反就输出“1”。我们使用任意的直线都无法完全分割该数据空间，而我们想要验证的就是 ReLU 这种“半截直线”的组合是否可以解决著名的“异或”问题，从而证明 ReLU 是否是一种非线性转换函数，也就证明了 ReLU 虽然外表“简单”，但同样拥有强大的非线性转换能力，换句话说，就是这家伙“城府深”。

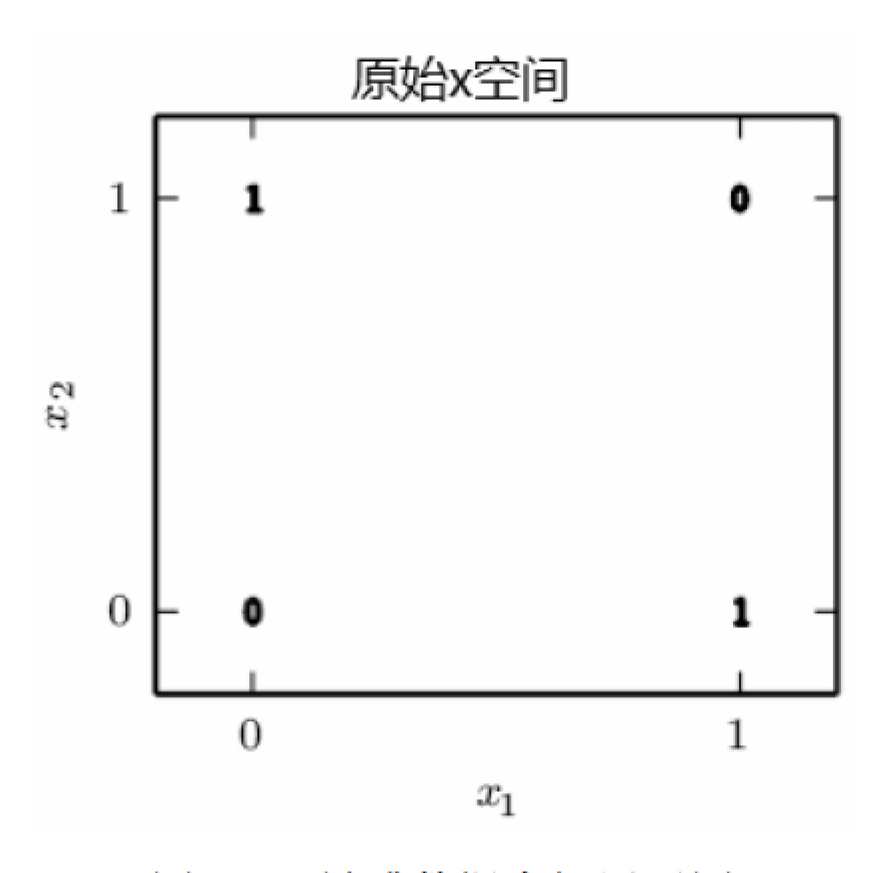


图 3-6　异或数据空间展示图

图 3-7 为我们设计用于解决异或问题的网络模型，该模型为一个标准的前馈神经网络，网络中每一个神经元都使用 ReLU 激活函数。式（3.8）为该网络的函数表达。其中矩阵 $\boldsymbol{W}$ 为第一层网络的连接权重，向量 $\boldsymbol{c}$ 为第一层的偏置项（也就是函数的常数项或截距），向量 $\boldsymbol{w}$ 为第二层的连接权重，向量 $\boldsymbol{b}$ 为第二层的偏置项。如式（3.9）所示，列出

了其中各连接的权值。

$$f(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c},\boldsymbol{w},b)=\boldsymbol{w}^{\top}\max\{0,\boldsymbol{W}^{\top}\boldsymbol{x}+\boldsymbol{c}\}+b \tag{3.8}$$

$$\boldsymbol{W}=\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix},\ \boldsymbol{c}=\begin{bmatrix}0\\ -1\end{bmatrix},\ \boldsymbol{w}=\begin{bmatrix}1\\ -2\end{bmatrix} \tag{3.9}$$

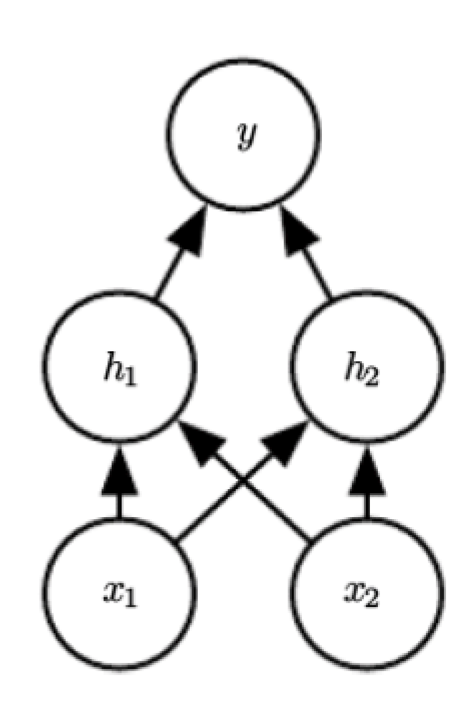


图 3-7　解决异或问题的网络模型

如式（3.10）所示，$\boldsymbol{X}$ 用于存放异或数据，$\boldsymbol{X}$ 中的每一行代表一条数据，每一列代表数据的维度。

$$\boldsymbol{X}=\begin{bmatrix}0 & 0\\ 0 & 1\\ 1 & 0\\ 1 & 1\end{bmatrix} \tag{3.10}$$

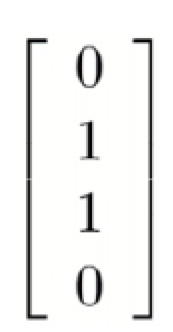
首先将数据的输入与对应的权重相乘再相加，也就是式（3.11），使用矩阵乘法。

$$\boldsymbol{XW}=\begin{bmatrix}0 & 0\\ 1 & 1\\ 1 & 1\\ 2 & 2\end{bmatrix} \tag{3.11}$$

接下来我们加上向量 $\boldsymbol{c}$，就得到如下左侧所示的结果，但我们的激活函数会将负数全部截断为 0，因此就得到如下右侧所示的最终结果。

$$\begin{bmatrix}0 & -1\\ 1 & 0\\ 1 & 0\\ 2 & 1\end{bmatrix}\quad\begin{bmatrix}0 & 0\\ 1 & 0\\ 1 & 0\\ 2 & 1\end{bmatrix}$$

我们将该结果用图 3-8 表示，就会发现一些有趣的现象，“异号”的数据被正确地映射到了同一点。那现在我们就可以轻松地找到一条直线，然后完美地分割这些数据。最后再使用向量 $\boldsymbol{w}$ 去乘特征转换后的数据，就得到了如下所示的最终结果，该结果是使用线性函数无法分割的异或数据问题。



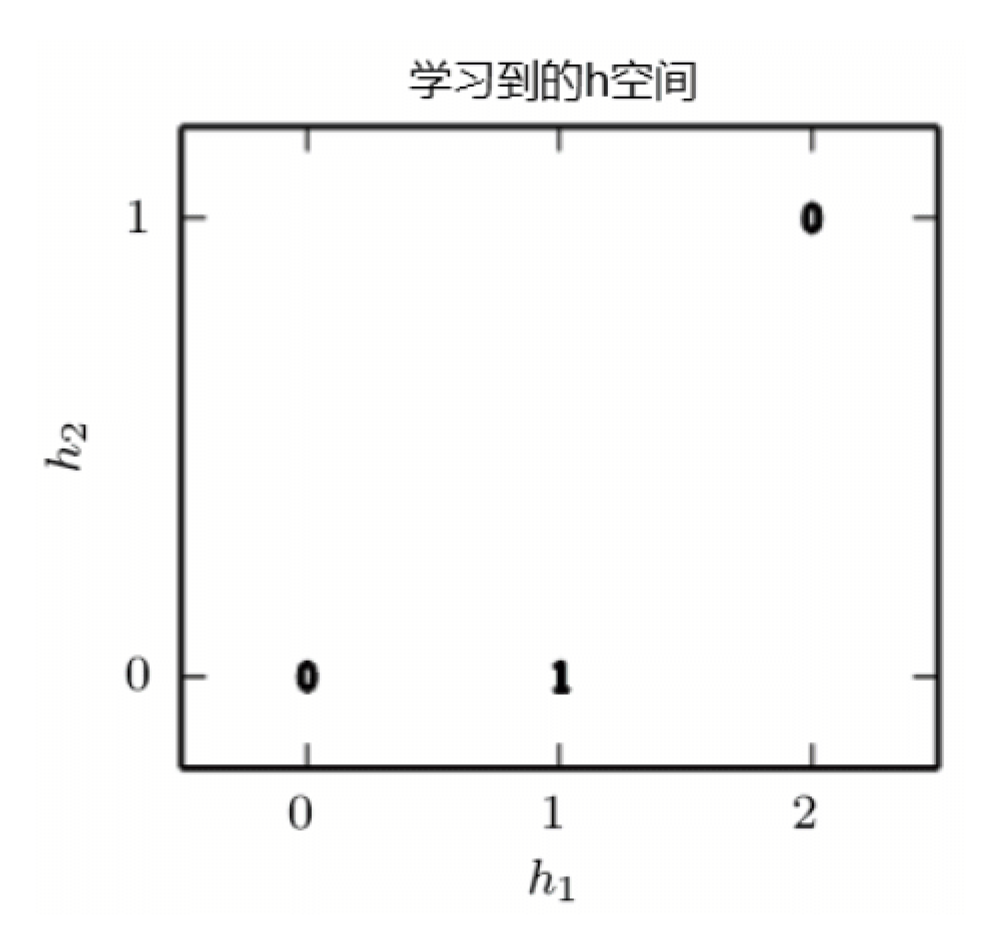


图 3-8　使用 ReLU 激活函数学习到的特征数据

目前为止，我们训练神经网络只有一招，那就是构造代价函数，然后求解梯度，修改网络权重。但使用 ReLU 函数时会有麻烦，因为 ReLU 在 0 点处不可导，该函数在零点的左导数为 0，而右导数为 1，当我们编程实现深度学习时，返回该激活函数其中一边的导数（如右导数）即可。不知对于比较“严谨”的读者来说，这算不算灾难，但请放松些，我们学的并不是数学。在实际应用中，虽然这些部分不可导，但其执行的性能依然足够好。

其中的部分原因在于，深度学习算法通常不是寻找代价函数最低点（训练数据的最优值不一定是测试数据的最优值），而是显著地降低代价函数。由于我们并不期望能到达梯度为 0 的点（山谷最低点），那么使用不严谨的梯度，如随机梯度下降法，也是可以接受的。ReLU 函数还有一个致命的缺点，那就是当神经元没有激活时，将永远无法修改其权重，就相当于神经元死亡了。

ReLU 可以看作是两条分段线性函数，只是其中一条分段函数输出恒为零。如式（3.12）所示，为通用的修正线性单元的函数表达式。

$$f(x,\alpha)=\max(0,x)+\alpha\min(0,x) \tag{3.12}$$

当 α=0 时，就是我们前面介绍的 ReLU。若固定 α= -1，该函数可表示为 $f(x)=|x|$，其就被称为**绝对值修正单元**（Absolute Value Rectification）[8]。而将 α 固定为较小的值，如 0.01 时，就称之为**裂缝修正单元**（Leaky ReLU）[9]。当 α 的值较小时，该函数的负半轴函数和横坐标的夹角就非常的小，因此使用“leaky”一词来形容，当然也可以使用 PReLU[10]将 α 作为一个可变参数来调整学习。

如果喜欢追求函数的表达能力，可以使用 **Maxout 单元**（Maxout Units）[11]。Maxout 由 k 段线性分段函数构成，具有很强的拟合能力。由 Maxout 组成的神经网络，不仅要学习神经元间的关系，还需要学习激活函数本身，这就好比一般的神经网络是一群比较“愚笨”的人去相互协作，而 Maxout 就好像一群比较“聪明”的人去互相帮助。Maxout 虽然能力强大，但

大量的参数也加重了训练的负担。

3.2 前馈神经网络

上文中用于解决“异或”问题的网络结构，其实就是一个标准的**前馈神经网络**（Feedforward Neural Networks，FNNs）或**多层感知机**（Multilayer Perceptrons，MLPs）[12]，如果更具体些，也可以称为浅层神经网络。所谓的“深”与“浅”其实是指输入数据到最后输出结果之间的层数，也称为**隐藏层**（hidden layer）。数据的输入也称为**输入层**，而结果的输出就称之为**输出层**，我们之所以称其为**前馈**（feedforward），是因为在执行阶段信息由数据 $\boldsymbol{X}$ 通过中间层函数 $f(x)$ 一层一层地计算，最终由输出层输出 y，信息不会反向传播，也不会横向传播。如图 3-9 所示的前馈神经网络示意图，红色的 L1 层神经元 $x1$，$x2$，$x3$ 表示输入数据，白色的 L4 层神经元表示输出单元，而灰色的 L2-L3 层神经元则表示隐藏单元。数据由输入层 L1 接收，然后通过 L2 和 L3 层进行特征转换，最后到达 L4 层输出结果。

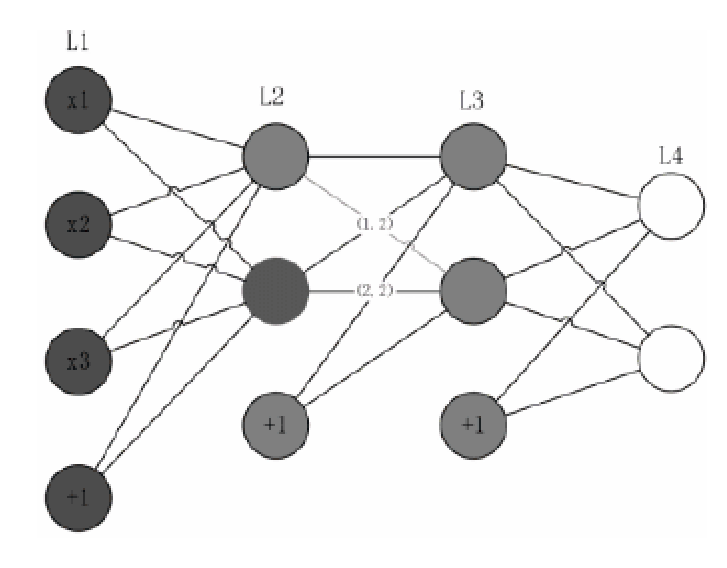


图 3-9　前馈神经网络示意图

前馈神经网络是一种极其重要的神经网络类型，甚至有时说到神经网络时，默认就是在说前馈神经网络。如今在计算机视觉领域大红大紫的**卷积神经网络**（Convolutional Neural Network，CNN）[13]就是一种特定的前馈神经网络，而在自然语言处理领域独占半壁江山的**循环神经网络**（Recurrent Neural Network，RNN）[1]其实也是踩着前馈网络前进的“一小步”。

3.2.1 输出层单元

神经网络的输出单元是高度与代价函数相关的，例如处理预测房价任务时，很可能就使用线性单元进行输出，如果处理分类任务，就使用 Sigmoid 单元或 Softmax 单元作为输出。神经网络大多数应用都是分类任务，但 Sigmoid 单元或 Softmax 单元又都有着容易饱和的情况，因此我们需要在其“犯错”时，给予较大的梯度去抗衡易饱和情况。由此，在神经网络中，默认使用**交叉熵**（Cross-entropy）作为代价函数。

3.2.2 隐藏层单元

神经网络隐藏层单元的设计是一个极度活跃的研究领域，并且还没有太多明确的理论原则。在 3.1.1 节提到过的所有神经元都可作为隐藏层单元使用，考虑到 Sigmoid 和 Tanh 单元存在易饱和现象，因此修正线性单元通常是一种非常优秀的默认隐藏层单元选择。但记住我们反复提到的没有免费午餐理论，具体何种任务适合何种神经元，需要大量的反复试验。比如在循环神经网络中，我们仍然不太常使用类似 ReLU 这样的分段线性激活函数，而更倾向于使用 Sigmoid 或 Tanh 神经元，即使我们知道其拥有易饱和的缺点。因此，隐藏层神经元的选择也是一种超参数，需要在实际应用中进行调试。

在前面的内容中，介绍了很多 ReLU 的优点，我们夸其为“平凡的天选之人”，当然也

介绍了其有些缺点，在 0 处不可导和负半轴无法训练。我们给出了一些 ReLU 的改进，比如**裂缝修正单元**（Leaky ReLU）和**绝对值修正单元**（Absolute Value Rectification），它们都是非常好的 ReLU 改进版本。当时我们说的是 ReLU 不可导也没关系，实践中还是很有用的。但毕竟“处女座”的读者也占十二分之一，我们还是需要“照顾”一下他们的情绪。其实也有平滑版本的修正线性单元，那就是 Softplus[14]，如式（3.13）所示为该函数的表达式。

$$\text{Softplus}(x)=\ln(1+e^{x}) \tag{3.13}$$

图 3-10 是 softplus 激活函数的图像，对比 Softplus 函数图像和 ReLU 函数图像，发现除了在 0 点处 Softplus 更平滑外，它们函数性质完全相同，但这其实有一点反常理，Glorot 在 2011 年比较了 Softplus 与 ReLU，并且发现后者的性能更优秀[15]。Softplus 通常是不鼓励使用的，虽然相比 ReLU，它可以处处可导，并且相比 Sigmoid 也没有易饱和性，但在实践中并不如我们想得那么美好。而我们在这里“跑偏似的”介绍 Softplus 是想说明，很多直觉上或理论上完美的东西，并不一定完美，实践才是检验真理的唯一标准。

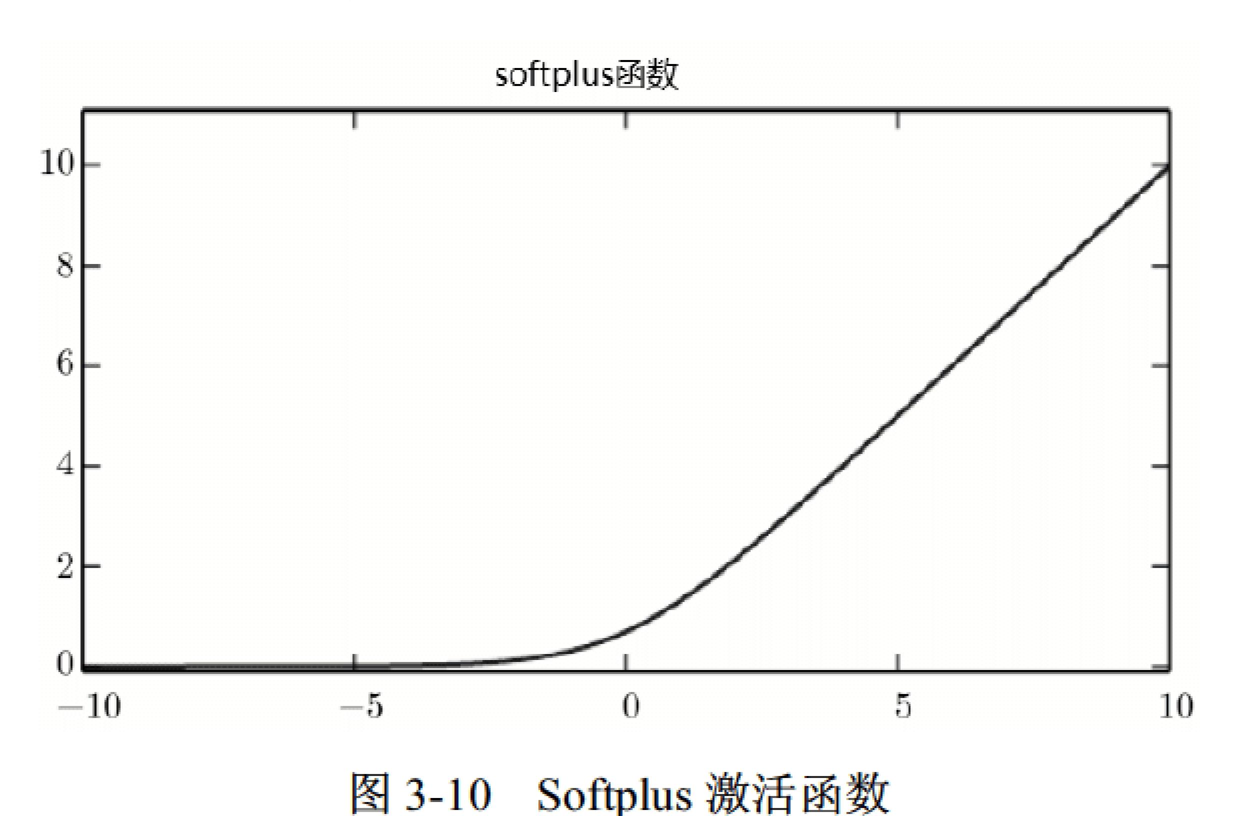


图 3-10　Softplus 激活函数

3.2.3　网络结构设计

神经网络结构设计是深度学习领域一个非常关键的部分，而结构设计主要讨论针对某项任务网络需要多少神经元以及这些神经元彼此如何连接。我们将会在本书第 6 章和第 7 章专门讨论卷积神经网络与循环神经网络这两类特殊的网络结构，而现在主要讨论两个话题：神经网络的**层数（深度）**以及每层神经元的**个数**。

通常情况下，层数越深的神经网络，其泛化效果也越好，但同时也越难优化。每层网络神经元个数越多，其能力也就越强，但也越容易过拟合。因此理想的网络结构需要通过反复的实验来获得。

通用逼近理论（Universal Approximation Theorem）[16]表明至少包含一个隐藏层的前馈神经网络，其在隐藏层使用“挤压式（squashing）”激活函数，如 Sigmoid 激活函数，只要**给予足够多的隐藏单元个数，其就可以逼近任意函数**。换而言之，就是只要隐藏层神经元足够多，训练数据的错误率就可以降到足够低。虽然这一理论给予了我们足够的“信心”：只要网络足够大，神经网络就可以**表示**任何我们想要获得的函数，但是这并不意味着我们能够**学习**到这种函数。

在浅层网络中（一隐藏层），一个隐藏单元可以当作是一个分割面，隐藏单元越多，其分割面也就越多，分割得也就越精细。这一过程可以想象成是将正方形切割成圆形，切割次数越多，正方形也就越像圆。而在极端情况下，我们为每一条数据都进行了“量身定制”的切割方式，但与其说是“学习”，还不如说是“记忆”。我们想要的是测试数据的规律，但却“记住”了训练数据，这也是神经网络面临的最大问题，易过拟合问题。但幸运的是，可以使用神经元的**层次组合**来降低神经元的数量。Montufar（2014）[17]研究显示，深层 ReLU 的能力需要一个指数级数量的浅层 ReLU 网络才能够表示。

总之，拥有一层隐藏单元的前馈神经网络就拥有了足够能力去表示任意的函数，但这种浅层网络神经元数量过于庞大，并且常常无法正确地学习，过拟合现象十分严重。在大多数情况下，使用**深层的神经网络可以有效地降低神经元的数量并同时降低泛化错误率**。如图 3-11 所示，为 Goodfellow 在识别图片数字任务中，显示的神经网络隐藏层层数与泛化正确率的关系图[18]，该结果表明随着网络层数增加，其泛化正确率也在不断增加。

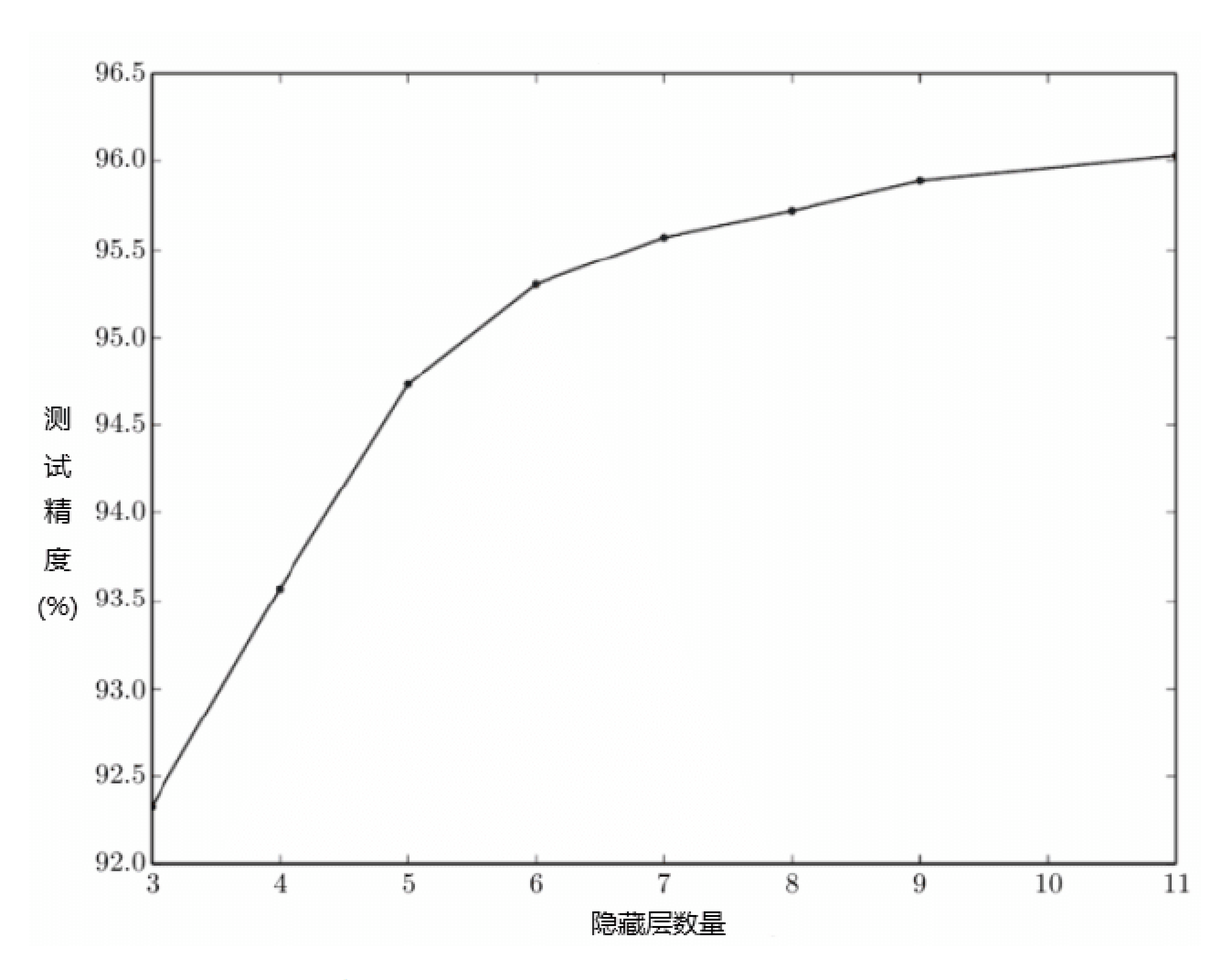


图 3-11　神经网络隐藏层层数与泛化正确率关系图

- 跃层连接

目前为止，我们描述的神经网络都是一层一层连接起来的，这很像政府组织结构，底层的信息往上一级传送，上一级整理分析信息后，再往其上级传送，依次传递到最高决策层。层次化传递最大的缺点就在于有用信息可能会丢失，并且还会带来额外的噪声，从而混淆了本已丢失的信息。那很自然地，高层为了获取“真实信息”也常常会派一些调查人员去查实，而底层也经常会出现“跨级汇报”的情况。如果平移到神经网络中，我们就可以设计一些跃层连接，比如某些第 2 层的神经元会连接到第 4 层甚至更高层，这样就缓解了信息丢失的情况，并且还减轻了梯度训练的困难。

3.3　BP 算法

接下来我们就详细介绍神经网络中最著名，并且也依然是现在深度学习中最具统治的算

法，**误差反向传播算法**（Back-Propagation Algorithm）[19]，简称 **BP 算法**。BP 算法在 20 世纪 70~80 年代被独立地发明了好几次，最早的版本要数 1969 年 Bryson 和 Ho 发明的线性版本[2]，而 Rumelhart、Williams 和 Hinton 在 1981 年发明了非线性 BP 算法[20]，不过第一次测试结果并不理想，所以他们就放弃了 BP 算法的使用。1986 年，他们使用一个令人信服的例子来说明该方法可以做的事情，并发表了论文，论文介绍了 BP 算法在学习多层非线性神经网络中有着非常良好的前景[19]。但在 20 世纪 90 年代后期，大多数严谨的研究者开始放弃 BP 算法，并且随着**支持向量机**（Support Vector Machine，SVM）[21]的发明，BP 算法被打入了冷宫，不过作为心理学模型来说，仍然被心理学家广泛的使用着。

BP 算法在 20 世纪 90 年代后期遭受冷落的主要原因在于当时计算机的**速度太慢**，有类标的**训练数据又太少**。并且当时的深度网络模型还太小，也没有合理的权重初始化策略，再加上没有重视梯度消失等原因，因此在深度网络上的 BP 算法效果并不好，这就阻止了 BP 算法的发展。而如今计算机性能，数据集都得到了有效提升，BP 算法再一次夺回了王座。

那什么又是 BP 算法呢？如果从数学角度出发，其实就是复合函数的求导问题。“幸运的”我们在中学时就被训练得身经百战了。当然，如果你忘记复合函数是什么了，我们也可以把 BP 算法简单地理解成一个“责任追究”的过程。我们把神经网络的前馈过程看作是某公司的战略决策过程，首先当然是基层员工去收集基础信息，然后各自整理分析；之后将各自信息再上报给其直接上级，然后该直接上级再一次汇总分析下属的信息；然后再将信息上报给上上级，这样一层一层分析上报，最后由该公司的 CEO 进行最后的决策。

那如果决策错误了又要怎么做呢？很自然地，那就是所有相关决策人进行反思与“责任追究”。首先是 CEO 进行自我的检讨与反思，反思后，他就要去追究相关的责任人，会指责其直接下属。而那些收到指责的下属便进行反思，然后再去指责他们的下属，然后就层层地往下传递这份“怨念”。需要注意的是，这里的“公司”比较特殊，一个下属会汇报信息给多个上级，而一个上级也会收到多份下属信息。因此，一个下属也会收到多个上级的指责，然后需要将所有上级的指责进行“叠加”。

接下来我们将详细地讲解 BP 算法的整个过程，会引入大量的公式与符号。这些都不难，但比较烦琐，希望读者能静下心来，留意每一个细节，特别是一些字母的上标与下标，希望读者保持清醒与耐心。

如图 3-12 所示，为 4 层神经网络的示意图，其中第 1 层 L1 为输入层，L4 为输出层，L2-L3 为隐藏层。我们使用 $w_{i,j}^{(l)}$ 表示第 l 层、第 i 神经元与第 l+1 层第 j 神经元的连接权重。如图 3-12 中红色的连接线为第 2 层的第 2 个神经元与第 3 层的第 2 个神经元的连接权重，因此用 $w_{2,2}^{(2)}$ 表示，图中浅蓝色的连接权重就表示为 $w_{1,2}^{(2)}$。我们使用 $b_j^{(l)}$ 表示第 l 层的偏置项连接到 l+1 层第 j 神经元。需要注意的是，在有些资料中下标的表示方法是相反的，读者阅读这些资料时需要细心与耐心。我们使用 $a_i^{(l)}$ 表示第 l 层、第 i 神经元的激活值（输出值），图中蓝色神经元的激活值就表示为 $a_2^{(2)}$。并且为了方便描述，当 l=1 时，将输入层的第 i 维输入特征同样用 a 表示，也就是 $a_i^{(1)} = x_i$。如式（3.14）所示，为 l+1 层第 j 神经元的激活表达式，其中函数 $f(x)$为激活函数。

$$a_j^{(l+1)} = f(\sum_{i=1}^{m} a_i^{(l)} w_{i,j}^{(l)} + b_j^{(l)}) \tag{3.14}$$

而图中蓝色神经元的激活值就可以表示为式（3.15）。

$$a_2^{(2)} = f(x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + x_3 w_{3,2}^{(1)} + b_2^{(1)}) \tag{3.15}$$

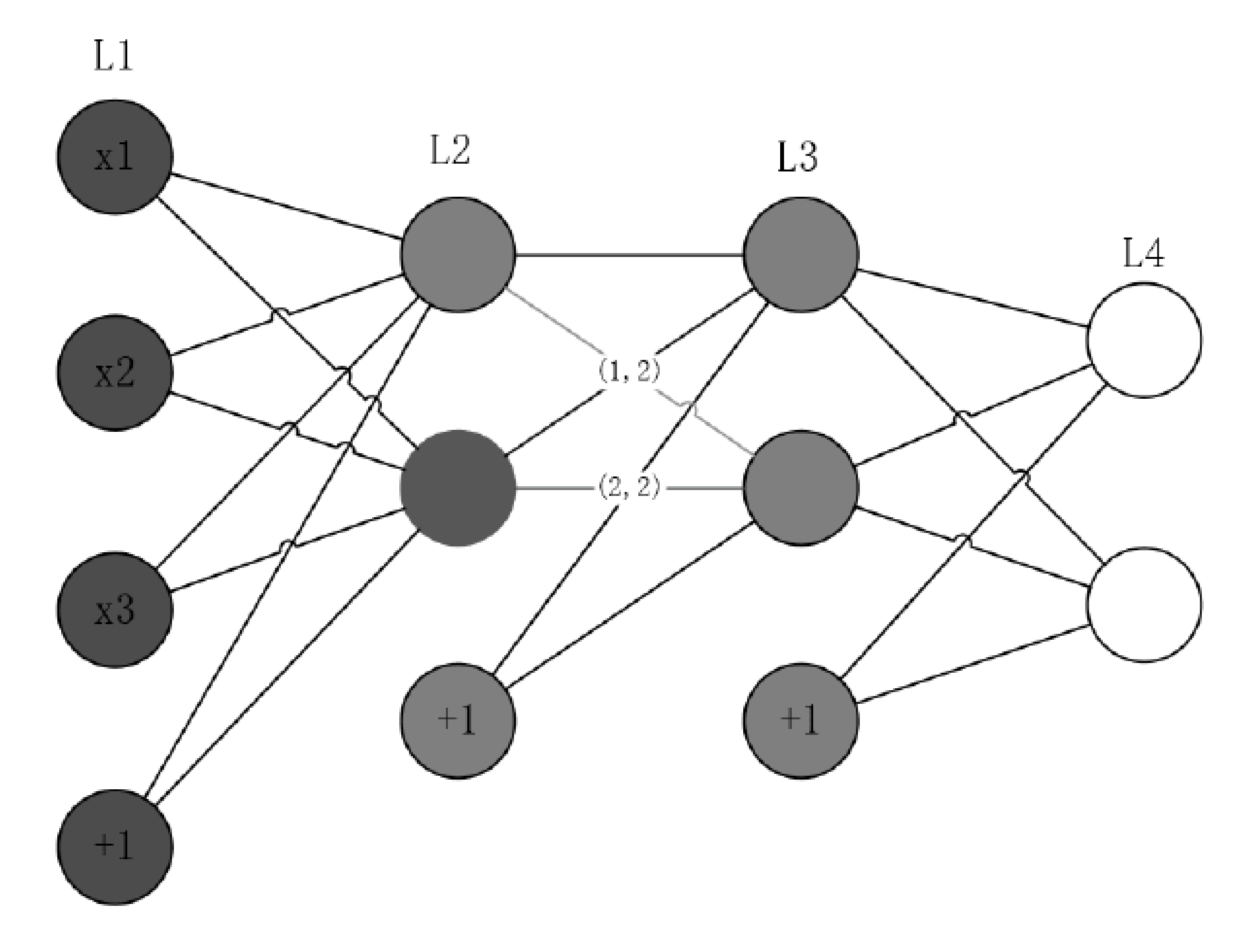


图 3-12　4 层神经网络示意图

以上的过程就为一个前馈传播的过程，那么如何修改每一处的权重呢？可以从最简单的开始，如图 3-13 所示，该神经网络是一个只有一维输入，每层只有一个神经元的特殊 4 层神经网络。

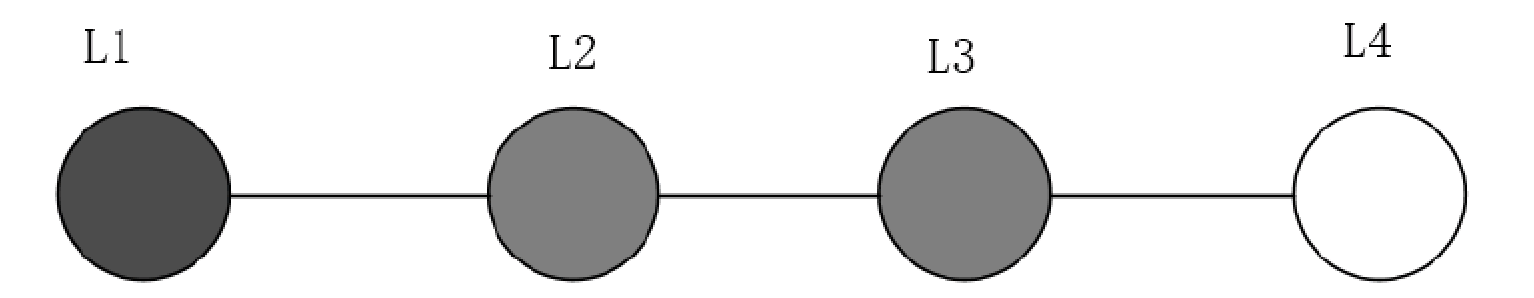


图 3-13　4 层简单神经网络示意图

该网络的输出为 output$=f(w_{1,1}^3 a_1^3)$，其中类标为 y，使用最小均方误差作为代价函数，其表达式就是式（3.16）。

$$J(w) = \frac{1}{2}(y - f(w_{1,1}^3 a_1^3))^2 \tag{3.16}$$

而我们的主要工作其实就是求解每一层权重的梯度，第 3 层连接到第 4 层的权重梯度就如式（3.17）所示，求解时将 w 作为自变量，a 作为系数。

$$\frac{\partial J(w)}{\partial w_{1,1}^3} = -(y - f(w_{1,1}^3 a_1^3)) f'(w_{1,1}^3 a_1^3) a_1^3 \tag{3.17}$$

那么第 2 层连接到第 3 层的权重梯度就如式（3.18）所示。

$$\frac{\partial J(w)}{\partial w_{1,1}^2} = -(y - f(w_{1,1}^3 a_1^3)) f'(w_{1,1}^3 a_1^3) w_{1,1}^3 f'(w_{1,1}^2 a_1^2) a_1^2 \tag{3.18}$$

第 1 层连接到第 2 层的权重梯度就如式（3.19）所示。

$$\frac{\partial J(w)}{\partial w_{1,1}^1} = -(y - f(w_{1,1}^3 a_1^3)) f'(w_{1,1}^3 a_1^3) w_{1,1}^3 f'(w_{1,1}^2 a_1^2) w_{1,1}^2 f'(w_{1,1}^1 x_1) x_1 \quad (3.19)$$

当你第一眼看到式（3.19），可能就会崩溃了，但如果自己动手求解一遍这个 4 层嵌套的复合函数，那可能会大有启发。

但这样做十分不方便，需要对其进行简化。现在我们再次从上层往下求梯度，但每层需要求解两种梯度，一种是该层的**权重梯度**，用 dw 表示；一种是该层的**输入梯度**，用 da 表示。在一些资料中输入梯度也被称为**残差**（error term），残差表明了该神经元对最终输出值的残差产生了多少影响，而最终输出节点的残差，我们直接使用网络产生的激活值与实际值之间的代价函数获得，对于隐藏节点的残差，其为该节点连接到上一层的所有残差加权求和。

从第 4 层，也就是从输出层开始，该层没有权重，仅仅计算输出层残差：

$$da_1^4 = -(y - f(w_{1,1}^3 a_1^3))$$

第 3 层中需要计算权重梯度 dw^3，该层残差 da^3。

$$dw_{1,1}^3 = da_1^4 \cdot f'(w_{1,1}^3 a_1^3) a_1^3$$

$$da_1^3 = da_1^4 \cdot f'(w_{1,1}^3 a_1^3) w_{1,1}^3$$

第 2 层中需要计算权重梯度 dw^2,该层残差 da^2。

$$dw_{1,1}^2 = da_1^3 \cdot f'(w_{1,1}^2 a_1^2) a_1^2$$

$$da_1^2 = da_1^3 \cdot f'(w_{1,1}^2 a_1^2) w_{1,1}^2$$

第 1 层中我们需要计算权重梯度 dw^1,该层残差 da^1。

$$dw_{1,1}^1 = da_1^2 \cdot f'(w_{1,1}^1 a_1^1) a_1^1$$

$$da_1^1 = da_1^2 \cdot f'(w_{1,1}^1 a_1^1) w_{1,1}^1$$

现在可以将这些结果，再代入到式（3.19）中进行验证。

我们将该递推公式进行推广，对于第 n 层（输出层）的每个输出单元 i，得到式（3.20）来计算残差。

$$da_i^n = J'(x; w) \quad (3.20)$$

对从第 2 层到第 n-1 层，第 l 层的第 i 节点的残差的计算，如式（3.21）所示。

$$da_i^l = \left(\sum_{j=1}^{m} da_j^{(l+1)} w_{i,j}^{(l)}\right) f'\left(\sum_{i=0}^{m} a_i^{(l-1)} w_{i,j}^{(l-1)}\right) \quad (3.21)$$

而第 l 层第 i 神经元到第 l+1 层第 j 神经元权重的偏导数如式（3.22）所示，第 l 层连接到第 l+1 层第 j 神经元的偏置项梯度如式（3.23）所示。

$$\frac{\partial J(w)}{\partial w_{i,j}^{l}} = da_j^{l+1} f'(\sum_{i=0}^{m} a_i^{(l)} w_{i,j}^{(l)}) a_i^l \tag{3.22}$$

$$\frac{\partial J(w)}{\partial b_j^{l}} = da_j^{l+1} f'(\sum_{i=0}^{m} a_i^{(l)} w_{i,j}^{(l)}) \tag{3.23}$$

为了确保自己深刻地理解，可以耐心地完成下面的练习。

BP 算法推倒练习：计算图 3-12 中各权重及残差
$da_1^4=1$ $da_2^4=1$
$da_1^3=$______
$da_2^3=$______
$dw_{1,1}^3=$______
$dw_{1,2}^3=$______
$dw_{2,1}^3=$______
$dw_{2,2}^3=$______
$da_1^2=$______
$da_2^2=(da_1^3 w_{1,2}^2 + da_2^3 w_{2,2}^2) f'(x_1 w_{1,2}^1 + x_2 w_{2,2}^1 + x_3 w_{3,2}^1 + b_2^1)$
$dw_{1,1}^2=$______
$dw_{1,2}^2=$______
$dw_{2,1}^2=$______
$dw_{2,2}^2=$______

3.4 深度学习编码实战上

本小节我们将编码完成全连接神经网络，并使用 CIFAR-10 数据集进行测试，在本章的练习中会逐步完成以下操作。

- 编码实现仿射层传播；
- 编码实现 ReLU 层传播；
- 编码实现组合单层神经元；
- 编码实现浅层全连接神经网络；
- 编码实现深层全连接神经网络。

接下来打开 DOS 窗口，将路径跳转到 DLAction 目录，启动 jupyter notebook，然后单击

“第 3 章练习-神经网络初步.ipynb”文件，按要求完成练习。

首先是相关的模块库导入，需要确认是否将“DLAction/classifiers/chapter3”模块导入成功。

库导入代码块：

```
# -*- coding: utf-8 -*-
import time
import numpy as np
import matplotlib.pyplot as plt
from classifiers.chapter3 import *
from utils import *
%matplotlib inline
plt.rcParams[ 'figure.figsize' ] = (10.0, 8.0 ) # 设置默认绘图尺寸。
plt.rcParams[ 'image.interpolation' ] = 'nearest'
plt.rcParams[ 'image.cmap' ] = 'gray'
%load_ext autoreload
%autoreload 2
def rel_error( x, y ) :
  # 计算相对错误。
  return np.max( np.abs( x – y ) / ( np.maximum( 1e-8, np.abs( x ) + np.abs( y ) ) ) )
```

模块导入成功后，我们载入 CIFAR-10 数据集。为了方便使用，我们将读取数据的各项预处理操作进行了单独地封装，存放在“DLAction/utils/data_utils.py”文件的 get_CIFAR10_data( ) 函数中。该函数的返回为数据字典，如训练数据存放在 data['X_train']中，验证数据存放在在 data['X_val']中。

CIFAR10 数据导入代码块：

```
# 将 CIFAR10 数据集的导入，切片，预处理操作进行封装。
data = get_CIFAR10_data( )
X_train =data[ 'X_train' ]
y_train =data[ 'y_train' ]
X_val = data[ 'X_val' ]
y_val = data[ 'y_val' ]
X_test = data[ 'X_test' ]
y_test = data[ 'y_test' ]
for k, v in data.iteritems( ):
  print '%s: ' % k, v.shape
```

执行上述代码块，期望得到下列结果。如下所示，导入的数据形状为原生的图像数据(数据个数，色道，宽，高)。

导入数据代码块结果：

```
X_val:   (1000L, 3L, 32L, 32L)
```

```
X_train:  (49000L, 3L, 32L, 32L)
X_test:  (1000L, 3L, 32L, 32L)
y_val:  (1000L,)
y_train:  (49000L,)
y_test:  (1000L,)
```

3.4.1 实现仿射传播

仿射（affine）传播就是将各个输入特征进行加权求和，如果下一层有 m 个神经元，那就相当于线性代数中的 m 组线性方程式。

需要注意：权重向量 $\boldsymbol{w}(D,M)$，D 表示输入层神经元数量，M 表示下一层神经元数量。这和我们的数据格式不相同，因此在计算前，需要将图像数据 $\boldsymbol{X}$(数据个数，色道，宽，高)，转化为(数据个数，色道×宽×高)。接下来我们就实现仿射层的前向传播，打开“DLAction/classifiers/chapter3/layers.py”文件，实现 affine_forward 函数。

affine_forward 函数代码块：

```
def affine_forward( x, w, b ) :
  """
  计算神经网络当前层的前馈传播，该方法计算在全连接情况下的得分函数。
  注意：如果不理解 affine 仿射变换，简单地理解为在全连接情况下的得分函数即可。
  输入数据 x 的形状为( N, d_1, ..., d_k )，其中 N 表示数据量，( d_1, ..., d_k )表示
  每一通道的数据维度，如果是图片，数据就为(长，宽，色道)。数据的总维度为
  D = d_1 * ... * d_k，我们需要将数据重塑成形状为(N, D)的数组再进行仿射变换。
  Inputs:
  - x: 输入数据，其形状为( N, d_1, ..., d_k )的 numpy 数组。
  - w: 权重矩阵，其形状为( D, M )的 numpy 数组，D 表示输入数据维度，
     M 表示输出数据维度，可以将 D 看成输入的神经元个数，M 看成输出神经元个数。
  - b: 偏置向量，其形状为( M, )的 numpy  数组。
  Returns  元组:
  - out: 形状为( N, M )的输出结果。
  - cache: ( x, w, b )将中间结果进行缓存便于反向传播使用。
  """
  out = None
  ####################################################################
  #                     任务：实现全连接前向传播。                     #
  #                  提示：首先你需要将输入数据重塑成行。              #
  ####################################################################
```

```
    ################################################################
    #                        结束编码                              #
    ################################################################
    cache = ( x, w, b )
    return out, cache
```

当完成了上述编码任务后，运行下面代码块进行测试，如果实现顺利，实现结果和正确结果的误差应该小于 1e-9。

测试 affine_forward 函数：

```
# np.linspaces，规定起点、终点（包含）、返回 array 的长度，
# 返回一个两端点间数值平均分布的 array。
num_inputs = 2
input_shape = ( 4, 5, 6 )
output_dim = 3
input_size = num_inputs * np.prod( input_shape )
weight_size = output_dim * np.prod( input_shape )
x = np.linspace( -0.1, 0.5, num = input_size ).reshape( num_inputs, *input_shape )
w = np.linspace( -0.2, 0.3, num = weight_size ).reshape( np.prod( input_shape ), output_dim )
b = np.linspace( -0.3, 0.1, num = output_dim )
out, _ = affine_forward( x, w, b )
correct_out = np.array( [ [ 1.49834967,   1.70660132,   1.91485297 ],
                          [ 3.25553199,   3.5141327,    3.77273342 ] ] )
# 比较实现的结果和正确结果，该误差应该小于 1e-9。
print '测试 affine_forward 函数:'
print '误差: ', rel_error( out, correct_out )
```

期望的测试结果：

```
测试 affine_forward 函数:
误差: 9.76984946819e-10
```

在前向传播中，实现的其实就是 out=$x1\times w1+x2\times w2+b$ 这个函数，那仿射层的反向传播，其实就是将 x，w，b 各自对应的梯度求出即可。需要注意的是，在缓存中我们存储的是原始的四维图像数据。因此，同前向传播一样，首先将数据转换为二维数组后，再进行求解。

仿射层反向传播：

```
def affine_backward( dout, cache ) :
    """
  计算仿射层的反向传播。
    Inputs:
    - dout: 形状为(N, M)的上层梯度。
```

```
- cache: 元组:
  - x: 形状为(N, d_1, ... d_k)的输入数据。
  - w: 形状为( D, M )的权重矩阵。
Returns 元组:
- dx: 输入数据 x 的梯度，其形状为(N, d1, ..., d_k)。
- dw: 权重矩阵 w 的梯度，其形状为( D, M )。
- db: 偏置项 b 的梯度，其形状为( M, )。
"""
x, w, b = cache
dx, dw, db = None, None, None
#############################################################################
#                        任务：实现仿射层反向传播                           #
#           注意：需要将 x 重塑成(N,D)后才能计算各梯度，                     #
#                  求完梯度后你需要将 dx 的形状与 x 重塑成一样。            #
#############################################################################




#############################################################################
#                              结束编码                                     #
#############################################################################
return dx, dw, db
```

当实现 affine_backward 函数后，接下来使用数值梯度检验实现是否准确。运行下列代码，相对梯度误差应该小于 1e-10。

affine_backward 函数梯度检验：

```
# 测试 affine_backward 函数。
from utils.gradient_check import *
x = np.random.randn( 10, 2, 3 )
w = np.random.randn( 6, 5 )
b = np.random.randn( 5 )
dout = np.random.randn( 10, 5 )
dx_num = eval_numerical_gradient_array( lambda x: affine_forward( x, w, b )[ 0 ], x, dout )
dw_num = eval_numerical_gradient_array( lambda w: affine_forward( x, w, b )[ 0 ], w, dout )
db_num = eval_numerical_gradient_array( lambda b: affine_forward( x, w, b )[ 0 ], b, dout )
_, cache = affine_forward( x, w, b )
dx, dw, db = affine_backward( dout, cache )
# 相对误差应该小于 1e-10。
```

```
print '测试 affine_backward 函数:'
print 'dx 误差: ', rel_error( dx_num, dx )
print 'dw 误差: ', rel_error( dw_num, dw )
print 'db 误差: ', rel_error( db_num, db )
```

正确测试结果：

```
测试 affine_backward 函数:
dx 误差:  1.29083036942e-09
dw 误差:  9.33833060677e-10
db 误差:  1.00945726594e-11
```

3.4.2　实现 ReLU 传播

我们介绍过很多激活函数，但其中效果最好，最普遍使用的是 ReLU 激活函数，并且其实现也非常简单。因此在默认情况下，都会使用 ReLU 作为隐藏层激活函数，接下来我们就来实现 ReLU 的前向传播以及反向传播过程。

ReLU 激活函数的公式为 $\max(0,x)$，我们直接使用 NumPy 提供的内置公式即可。打开“DLAction/classifiers/chapter3/layers.py”文件，实现 relu_forward 函数。

relu_forward 代码块：

```
def relu_forward( x ):
    """
    计算 ReLUs 激活函数的前向传播，并保存相应缓存。
    Input:
    - x: 输入数据。
    Returns 元组：
    - out: 和输入数据 x 形状相同。
    - cache: x。
    """
    out = None
    ##########################################################################
    #                        任务：实现 ReLU 的前向传播。                        #
    #                      注意：只需要一行代码即可完成。                        #
    ##########################################################################


    ##########################################################################
    #                               结束编码                                   #
    ##########################################################################
    cache = x
    return out, cache
```

完成 relu_forward 函数后，使用下列代码块进行验证，其误差大约为 1e-8。

relu_forward 函数验证代码块：

```
# 测试 relu_forward 函数。
x = np.linspace( -0.5, 0.5, num = 12 ).reshape( 3, 4 )
out, _ = relu_forward( x )
correct_out = np.array( [ [ 0.,          0.,          0.,          0.,          ],
                          [ 0.,          0.,          0.04545455,  0.13636364,],
                          [ 0.22727273,  0.31818182,  0.40909091,  0.5,         ] ] )
# 比较输出结果，其误差大约为 1e-8。
print '测试 relu_forward 函数:'
print '误差: ', rel_error( out, correct_out )
```

正确测试结果：

```
测试 relu_forward 函数:
误差:  4.99999979802e-08
```

完成简单的 ReLU 前向传播后，接下来实现其反向传播。在前馈过程中，我们简单地执行 max(0,x)函数，而反向传播时也非常简单，我们只需要将 x>0 处的梯度保留，将 x<=0 处的梯度，设置为 0 即可。

relu_backward 代码块：

```
def relu_backward( dout, cache ):
  """
  计算 ReLUs 激活函数的反向传播。
  Input:
  - dout: 上层误差梯度。
  - cache: 输入数据 x，其形状应该和 dout 相同。
  Returns:
  - dx: x 的梯度。
  """
  dx, x = None, cache
  ##########################################################################
  #                  任务：实现 ReLU 反向传播。                            #
  ##########################################################################



  ##########################################################################
  #                             结束编码                                   #
  ##########################################################################
  return dx
```

实现 relu_backward 函数后，运行下列代码块进行梯度检验，其相对误差大约为 1e-12。

relu_backward 函数梯度检验代码块：

```
x = np.random.randn( 10, 10 )
dout = np.random.randn( *x.shape )
dx_num = eval_numerical_gradient_array( lambda x: relu_forward( x )[ 0 ], x, dout )
_, cache = relu_forward( x )
dx = relu_backward( dout, cache )
# 其相对误差大约为 1e-12。
print '测试 relu_backward 函数:'
print 'dx 误差: ', rel_error( dx_num, dx )
```

正确的测试结果为：

```
测试 relu_backward 函数:
dx 误差:  3.27562024188e-12
```

3.4.3　组合单层神经元

接下来我们将上述的 affine 传播和 ReLU 传播组合在一起，形成一层完整的神经元。在前向传播时，由于 affine 和 ReLU 各自缓存都需要在反向传播时使用，因此需要将这些缓存都保存起来，代码块如下所示。

affine_relu_forward 函数代码块：

```
def affine_relu_forward( x, w, b ):
  """
   ReLU 神经元前向传播。
  Inputs:
  - x: 输入到 affine 层的数据。
  - w, b:  affine 层的权重矩阵和偏置向量。
  Returns 元组:
  - out: ReLU 的输出结果。
  - cache: 前向传播的缓存。
  """
  #######################################################################
  #              任务：实现 ReLU 神经元前向传播。                       #
  #          注意：需要调用 affine_forward 以及 relu_forward 函数，     #
  #              并将各自的缓存保存在 cache 中。                        #
  #######################################################################



  #######################################################################
```

```
    #                          结束编码                          #
    ######################################################################
    return out, cache
```

affine_relu_backward 函数代码块：

```
def affine_relu_backward( dout, cache ):
    """
     ReLU 神经元的反向传播。
    Input:
    - dout: 上层误差梯度。
    - cache: affine 缓存及 relu 缓存。
    Returns:
    - dx: 输入数据 x 的梯度。
    - dw: 权重矩阵 w 的梯度。
    - db: 偏置向量 b 的梯度。
    """
    ##################################################################
    #                任务：实现 ReLU 神经元反向传播。                #
    ##################################################################




    ###################################################################
    #                        结束编码                                 #
    ###################################################################
    return dx, dw, db
```

实现 affine_relu_forward 和 affine_relu_backward 函数之后，运行下列代码进行梯度检验。

affine_relu 梯度检验代码块：

```
# 初始化。
x = np.random.randn( 2, 3, 4 )
w = np.random.randn( 12, 10 )
b = np.random.randn( 10 )
dout = np.random.randn( 2, 10 )
# 执行 ReLU，获取分析梯度。
out, cache = affine_relu_forward( x, w, b )
dx, dw, db = affine_relu_backward( dout, cache )
# 获取数值梯度。
dx_num = eval_numerical_gradient_array( lambda x: affine_relu_forward( x, w, b )[ 0 ], x, dout )
```

```
dw_num = eval_numerical_gradient_array( lambda w: affine_relu_forward( x, w, b)[ 0 ], w, dout )
db_num = eval_numerical_gradient_array( lambda b: affine_relu_forward( x, w, b )[ 0 ], b, dout )
# 比较相对误差。
print '测试 ReLU 神经元相对误差:'
print 'dx 误差: ', rel_error( dx_num, dx )
print 'dw 误差: ', rel_error( dw_num, dw )
print 'db 误差: ', rel_error( db_num, db )
```

正确的测试结果：

```
测试 ReLU 神经元相对误差:
dx 误差:   1.80292854065e-10
dw 误差:   3.15495388158e-10
db 误差:   4.91876024003e-12
```

- 输出层：Softmax

在上一章节中，我们已经完美地实现了 Softmax 分类器，现在我们将其存放在“DLAction/classifiers/chapter3/layers.py”文件中，直接使用即可。阅读下列代码，确保你已经完全掌握了 Softmax。

softmax_loss 函数代码块：

```
def softmax_loss( x, y ):
    probs = np.exp ( x - np.max( x, axis = 1, keepdims = True ) )
    probs / = np.sum( probs, axis = 1, keepdims = True )
    N = x.shape[ 0 ]
    loss = -np.sum( np.log( probs[ np.arange( N ), y ] ) ) / N
    dx = probs.copy( )
    dx[ np.arange( N ), y ] -= 1
    dx / = N
    return loss, dx
```

运行下列代码以确认我们的实现是正确的。

Softmax 验证代码块：

```
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn( num_inputs, num_classes )
y = np.random.randint( num_classes, size = num_inputs )
dx_num = eval_numerical_gradient( lambda x: softmax_loss( x, y )[ 0 ],
  x, verbose = False )
loss, dx = softmax_loss( x, y )
# 测试 softmax_loss 函数。损失值大约为 2.3，dx 误差大约为 1e-8。
print '\n 测试 softmax_loss:'
```

```
print 'loss: ', loss
print 'dx error: ', rel_error( dx_num, dx )
```

正确的测试结果：

```
测试 softmax_loss:
loss:   2.30251623362
dx error:   8.01877448098e-09
```

3.4.4 实现浅层神经网络

完成上述的模块后，接下来我们将它们组装起来，编码实现浅层（单隐藏层）全连接神经网络。打开“DLAction/classifiers/chapter3/shallow_layer_net.py”文件，阅读相关内容完成相应任务。在浅层全连接神经网络中，隐藏层使用 ReLU 作为激活函数，输出层使用 softmax 作为分类器，该网络结构应该为：affine-relu-affine–softmax。

- 网络初始化

在神经网络初始化时，我们需要将连接权重初始化为高斯分布，其均值为零，权重的取值范围（标准差）作为超参数控制，偏置项通常设置为零，代码块如下所示。

shallow_layer_net.init 代码块：

```
def __init__( self, input_dim = 3 * 32 * 32, hidden_dim = 100, num_classes = 10,
                    weight_scale = 1e-3, reg = 0.0 ):
        """
        初始单隐藏层全连接神经网络。
        Inputs:
        - input_dim: 输入数据维度。
        - hidden_dim: 隐藏层维度。
        - num_classes: 分类数量。
        - weight_scale: 权重取值范围，给予初始化权重的标准差。
        - reg: L2 正则化的权重衰减系数。
        """
        self.params = { }
        self.reg = reg
        ############################################################################
        #                          任务：初始化权重以及偏置项。                        #
        # 权重应该服从标准差为 weight_scale 的高斯分布，偏置项应该初始化为 0，          #
        #   所有权重矩阵和偏置向量应该存放在 self.params 字典中。                     #
        #   第一层的权重和偏置使用键值 'W1'以及'b1'，第二层使用'W2'以及'b2'。           #
        ############################################################################
```

```
        ########################################################################
        #                              结束编码                                #
        ########################################################################
```

- 网络损失函数

浅层神经网络的损失函数是网络两个执行阶段的融合，当处于测试阶段时，我们仅仅输入数据，不输入类标，函数返回 Softmax 层中的得分函数即可；当处于训练阶段时，需要计算数据的损失值以及相应的权重梯度，代码块如下所示。

shallow_layer_net.loss 代码块：

```
def loss( self, X, y = None ):
        """
        计算数据 X 的损失值以及梯度。
        Inputs:
        - X: 输入数据，形状为( N, d_1, ..., d_k )的 numpy 数组。
        - y: 数据类标，形状为( N, )的 numpy 数组。
        Returns:
        如果 y 为 None，表明网络处于测试阶段直接返回输出层的得分即可：
        - scores: 形状为( N, C )，其中 scores[ i, c ]是数据 X[ i ]在第 c 类上的得分。
        如果 y 为 not None，表明网络处于训练阶段，返回一个元组：
        - loss: 数据的损失值。
        - grads: 与参数字典相同的梯度字典，键值和参数字典的键值相同。
        """
        scores = None
        ############################################################
        #              任务：实现浅层网络的前向传播过程，          #
        #                      计算各数据的分类得分。              #
        ############################################################



        ############################################################
        #                         结束编码                         #
        ############################################################
        # 如果 y 为 None 直接返回得分。
        if y is None:
          return scores
        loss, grads = 0, { }
```

```
################################################################
#                    任务：实现浅层网络的反向传播过程，                    #
#          将损失值存储在 loss 中，将各层梯度存储在 grads 字典中。       #
#                注意：别忘了还要计算权重衰减。                      #
################################################################





############################################################
#                          结束编码                         #
############################################################
return loss, grads
```

完成上述编码后，我们使用下列代码块检验实现，如果出现异常，将会输出相应的异常结果。得分函数误差应该小于 1e-6，损失值误差应该小于 1e-10，梯度误差都应该在 1e-7 以内。

浅层网络测试代码块：

```
N, D, H, C = 3, 5, 50, 7
X = np.random.randn( N, D )
y = np.random.randint( C, size = N )
std = 1e-2
model = ShallowLayerNet( input_dim = D, hidden_dim = H,
num_classes = C, weight_scale = std )
print '测试初始化 ... '
W1_std = abs( model.params[ 'W1' ].std( ) – std )
b1 = model.params[ 'b1' ]
W2_std = abs( model.params[ 'W2' ].std( ) – std )
b2 = model.params[ 'b2' ]
assert W1_std < std / 10, '第一层权重初始化有问题'
assert np.all( b1 == 0 ), '第一层偏置初始化有问题'
assert W2_std < std / 10, '第二层权重初始化有问题'
assert np.all( b2 == 0 ), '第二层偏置初始化有问题
print '测试前向传播过程 ... '
model.params[ 'W1' ] = np.linspace( -0.7, 0.3, num = D * H ).reshape( D, H )
model.params[ 'b1' ] = np.linspace( -0.1, 0.9, num = H )
model.params[ 'W2' ] = np.linspace( -0.3, 0.4, num = H * C ).reshape( H, C )
model.params[ 'b2' ] = np.linspace( -0.9, 0.1, num = C )
X = np.linspace( -5.5, 4.5, num = N * D ).reshape( D, N ).T
```

```
scores = model.loss( X )
correct_scores = np.asarray(
  [ [ 11.53165108,   12.2917344,    13.05181771,   13.81190102,
     14.57198434, 15.33206765,   16.09215096 ],
   [ 12.05769098,   12.74614105,   13.43459113,   14.1230412,
     14.81149128, 15.49994135,   16.18839143 ],
   [ 12.58373087,   13.20054771,   13.81736455,   14.43418138,
      15.05099822, 15.66781506,   16.2846319 ] ] )
scores_diff = np.abs( scores - correct_scores ).sum( )
assert scores_diff < 1e-6, '前向传播有问题'
print '测试训练损失(无正则化)'
y = np.asarray( [ 0, 5, 1 ] )
loss, grads = model.loss( X, y )
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, '训练阶段的损失值(无正则化)有问题'
print '测试训练损失(正则化 0.1)'
model.reg = 1.0
loss, grads = model.loss( X, y )
correct_loss = 26.5948426952
assert abs( loss - correct_loss ) < 1e-10, '训练阶段的损失值(有正则化)有问题'
for reg in [ 0.0, 0.7 ]:
  print '梯度检验，正则化系数 =', reg
  model.reg = reg
  loss, grads = model.loss( X, y )
  for name in sorted( grads ):
    f = lambda _: model.loss( X, y )[ 0 ]
grad_num = eval_numerical_gradient( f, model.params[ name ],
 verbose = False )
    print '%s  相对误差: %.2e' % ( name, rel_error(grad_num, grads[ name ] ) )
```

正确编码后的测试结果：

测试初始化 … 测试前向传播过程 … 测试训练损失(无正则化) 测试训练损失(正则化 0.1)	梯度检验， 正则化系数 = 0.0 W1 相对误差: 1.83e-08 W2 相对误差: 3.48e-10 b1 相对误差: 6.55e-09 b2 相对误差: 4.33e-10	梯度检验， 正则化系数 = 0.7 W1 相对误差: 2.53e-07 W2 相对误差: 2.85e-08 b1 相对误差: 1.35e-08 b2 相对误差: 9.09e-10

成功完成上述任务后，接下来我们完成训练浅层全连接网络的函数编码，该流程和 2.6.5 节中的 Softmax 训练过程一样，你应该非常熟悉了，这里就不详细解释了。阅读 ShallowLayerNet 类的 train()及 predict()函数，确保自己确实了解整个训练流程。执行下列代

码块，测试网络性能。

浅层神经网络训练代码块：

```
input_size = 32 * 32 * 3
hidden_size = 100
num_classes = 10
net = ShallowLayerNet( input_size, hidden_size, num_classes )
# 训练网络。
stats = net.train( X_train, y_train, X_val, y_val,
                num_iters = 2000, batch_size = 500,
                learning_rate = 1e-3, learning_rate_decay = 0.95,
                reg = 0.6, verbose = True )
# 验证结果。
val_acc = ( net.predict( X_val ) == y_val ).mean( )
print '最终验证正确率: ', val_acc
print '历史最佳验证正确率: ', stats[ 'best_val_acc' ]
```

测试结果：

```
迭代次数  0 / 2000:  损失值  2.397091
迭代次数  100 / 2000:  损失值  1.777225
迭代次数  200 / 2000:  损失值  1.672178
迭代次数  300 / 2000:  损失值  1.694989
                ………
迭代次数  1700 / 2000:  损失值  1.401761
迭代次数  1800 / 2000:  损失值  1.352970
迭代次数  1900 / 2000:  损失值  1.459668
最终验证正确率:   0.502
历史最佳验证正确率:   0.522
```

非常令人激动，我们的训练结果相比于 Softmax，提升了很大的性能，接下来可视化整个训练过程，代码块如下所示，可视化结果如图 3-14 所示。

可视化训练历史代码块：

```
# 绘制损失函数变化曲线。
plt.subplot( 2, 1, 1 )
plt.plot( stats[ 'loss_history' ] )
plt.title( 'Loss history' )
plt.xlabel( 'Iteration' )
plt.ylabel( 'Loss' )
plt.subplot( 2, 1, 2 )
plt.plot( stats[ 'train_acc_history' ], label = 'train' )
plt.plot( stats[ 'val_acc_history' ], label = 'val' )
```

```
plt.plot( [ 0.5 ] * len( stats[ 'val_acc_history' ] ), 'k--' )
plt.title( 'Classification accuracy history' )
plt.xlabel( 'Epoch' )
plt.ylabel( 'Clasification accuracy' )
plt.show( )
```

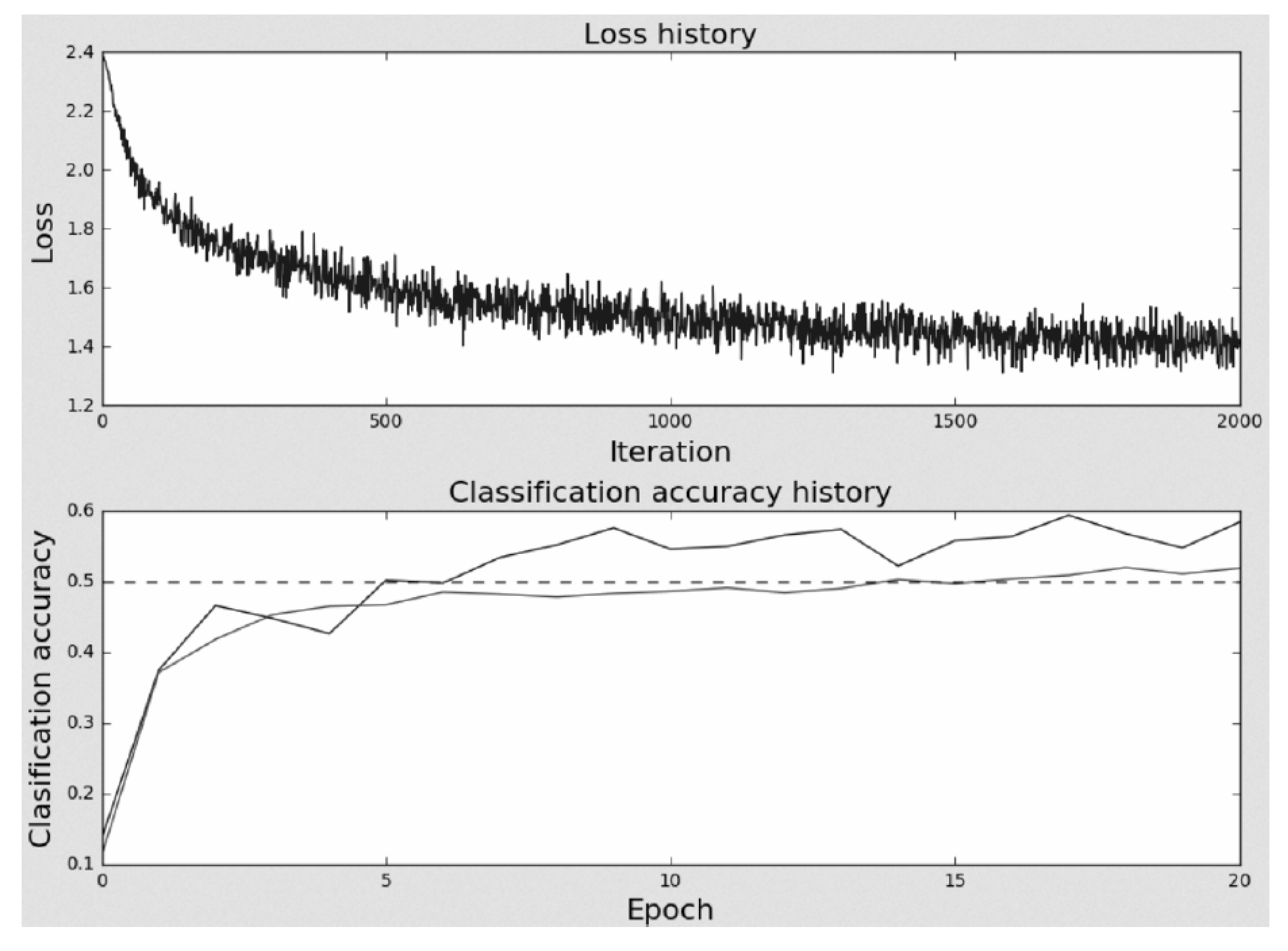


图 3-14　浅层神经网络训练曲线

虽然浅层神经网络的性能已经超过了 50%，相比于单独的 Softmax 分类器有了很大的性能提升，但从图 3-14 的训练曲线可以看出，该网络并没有出现明显的过拟合现象，并且验证精度也没有下降趋势。因此现在需要修改其超参数配置，尽最大的努力训练出一个最佳的浅层神经网络，祝你好运！

3.4.5　实现深层全连接网络

深层神经网络相比于浅层神经网络，虽然仅仅是隐藏层数量变多而已，但它却具有了浅层网络没有的强大能力。由于我们已经实现了浅层网络，深层网络的实现将变得非常简单。现在要做的仅仅是将浅层网络中固定的单隐藏层变成任意多层即可。在深层全连接神经网络，隐藏层使用 ReLU 作为激活函数，输出层使用 Softmax 作为分类器。

打开“DLAction/classifiers/chapter3/fc_net.py”文件，该网络结构应该为{ affine – relu } x (L -1) - affine - softmax。我们已经将每层的维度存储在 layers_dims = [input_dim] + hidden_dims + [num_classes]中了，直接使用即可。仅仅需要添加一个循环即可，需要注意循环是从 0 开始计数的，而我们的权重参数是从 1 开始计数。

初始化深层网络权重代码块：

```
def __init__( self, input_dim = 3 * 32 * 32, hidden_dims = [ 50, 50 ],
num_classes = 10,    reg = 0.0, weight_scale = 1e-3 ):
        """
        深层神经网络初始化。
        Inputs:
        - input_dim: 输入数据维度。
        - hidden_dim: 隐藏层各层维度。
        - num_classes: 分类数量。
        - weight_scale: 权重取值范围，初始化权重的标准差。
        - reg: L2 正则化的权重衰减系数。
        """
        self.reg = reg
        self.num_layers = 1 + len( hidden_dims )
        self.params = { }
        # 这里存储的是每层的神经元数量。
        layers_dims = [ input_dim ] + hidden_dims + [ num_classes ]
        ############################################################################
        #                   任务：初始化任意多层权重以及偏置项。                   #
        # 权重应该服从标准差为 weight_scale 的高斯分布，偏置项应该初始化为 0，    #
        # 所有权重矩阵和偏置向量应该存放在 self.params 字典中。                    #
        # 第一层的权重和偏置使用键值'W1'以及'b1'，第 n 层使用'Wn'以及'bn'。        #
        ############################################################################




        ############################################################################
        #                             结束编码                                     #
        ############################################################################
```

和浅层网络类似，接下来我们将完成深层网络的损失函数代码块，如下列代码所示，注意正则化因子的添加。**提示：**网络前 n-1 使用 affine_relu_forward，最后一层使用 affine_forward 执行前向传播过程。计算梯度时，情况类似，顺序相反。

深层网络损失函数代码块：

```
    def loss( self, X, y = None ) :
        """
        计算数据 X 的损失值以及梯度。
        Inputs:
```

```
- X: 输入数据，形状为(N, d_1, ..., d_k)的 numpy 数组。
- y: 数据类标，形状为( N, )的 numpy 数组。
Returns:
如果 y 为 None，表明网络处于测试阶段直接返回输出层的得分即可。
- scores:形状为(N, C)，其中 scores[ i, c] 是数据 X[ i ] 在第 c 类上的得分。
如果 y 为 not None，表明网络处于训练阶段，返回一个元组：
- loss:数据的损失值。
- grads:与参数字典相同的梯度字典，键值和参数字典的键值要相同。
"""
scores = None
##################################################################
#                    任务：实现深层网络的前向传播过程，                    #
#                          计算各数据的分类得分。                          #
##################################################################




################################################################
#                                结束编码                             #
################################################################
loss, grads = 0.0, { }
#################################################################
#                    任务：实现深层网络的反向传播过程，                   #
#          将损失值存储在 loss 中，将各层梯度存储在 grads 字典中。          #
#                    注意：别忘了还要计算权重衰减。                       #
##############################################################




#################################################################
#                                结束编码                             #
################################################################
return loss, grads
```

在深层网络的 train 函数中，我们已经实现了绝大部分内容，现在只需要调用 loss 函数计算梯度，然后结合学习率，修改权重即可。

深层神经网络 train 函数代码块：

```
def train( self, X, y, X_val, y_val, learning_rate = 1e-3, learning_rate_decay = 0.95,
           num_iters = 100, batch_size = 200, verbose = False ) :
    """
    使用随机梯度下降训练神经网络。
    Inputs:
    - X: 训练数据。
    - y: 训练数据类标。
    - X_val: 验证数据。
    - y_val:验证数据类标。
    - learning_rate: 学习率。
    - learning_rate_decay: 学习率衰减系数。
    - reg: 权重衰减系数。
    - num_iters: 迭代次数。
    - batch_size: 批量大小。
    - verbose:是否在训练过程中打印结果。
    """
    num_train = X.shape[ 0 ]
    iterations_per_epoch = max( num_train / batch_size, 1)
    loss_history = [ ]
    train_acc_history = [ ]
    val_acc_history = [ ]
    best_val = -1
    for it in xrange( num_iters ) :
      X_batch = None
      y_batch = None
      sample_index = np.random.choice( num_train, batch_size, replace=True)
      X_batch = X [ sample_index, : ]   # ( batch_size, D )。
      y_batch = y[ sample_index ]   # (1, batch_size )。
      # 计算损失以及梯度。
      loss, grads = self.loss( X_batch, y = y_batch )
      loss_history.append( loss )
      # 修改权重。
      ############################################################
      #                    任务：修改深层网络的权重。              #
      ############################################################


      ############################################################
```

```
    #                        结束编码                    #
    ############################################################
    if verbose and it % 100 == 0:
      print 'iteration %d / %d: loss %f' % ( it, num_iters, loss )
    if it % iterations_per_epoch == 0:
      # 检验精度。
      train_acc = ( self.predict( X ) == y ).mean( )
      val_acc = ( self.predict( X_val ) == y_val ).mean( )
      train_acc_history.append( train_acc )
      val_acc_history.append( val_acc )
      if ( best_val < val_acc ):
          best_val = val_acc
      # 学习率衰减。
      learning_rate * = learning_rate_decay
  return {
    'loss_history': loss_history,
    'train_acc_history': train_acc_history,
    'val_acc_history': val_acc_history,
    'best_val_acc':best_val
  }
```

深层网络的预测也十分简单，仅仅返回前向传播的最高得分的位置即可，代码块如下所示。

深层网络 predict 函数代码块：

```
def predict( self, X ) :
    """
    Inputs:
    - X：输入数据。
    Returns:
    - y_pred：预测类别。
    """
    y_pred = None
    ############################################################
    #          任务：执行深层网络的前向传播，                  #
    #          然后使用输出层得分函数预测数据类标。            #
    ############################################################
```

```
    ########################################################
    #                   结束编码                            #
    ########################################################
    return y_pred
```

完成深层神经网络的编码后，接下来我们就测试编码是否正确，运行下列代码块。

测试深层网络代码块：

```
N, D, H1, H2, H3, C = 2, 15, 20, 30, 20, 10
X = np.random.randn( N, D )
y = np.random.randint( C, size = ( N, ) )
for reg in [ 0, 0.11, 3.14 ] :
  print '权重衰减系数=', reg
  model = FullyConnectedNet( input_dim = D, hidden_dims = [ H1, H2, H3 ],
 num_classes = C, reg = reg, weight_scale = 5e-2 )
  loss, grads = model.loss( X, y )
  print '初始化化损失值: ', loss
  for name in sorted( grads ) :
      f = lambda _: model.loss( X, y )[ 0 ]
grad_num = eval_numerical_gradient( f, model.params[ name ],
verbose = False, h =1e-5)
    print '%s  相对误差: %.2e' % ( name, rel_error( grad_num, grads[ name ] ) )
```

希望能得到类似以下的测试结果。

深层网络测试的正确结果：

权重衰减系数 = 0	权重衰减系数 = 0.11	权重衰减系数 = 3.14
初始化化损失值:	初始化化损失值:	初始化化损失值:
2.30224308129	2.54242996031	8.8189968312
W1 相对误差: 4.30e-07	W1 相对误差: 4.92e-07	W1 相对误差: 1.28e-08
W2 相对误差: 3.45e-06	W2 相对误差: 1.04e-06	W2 相对误差: 1.33e-07
W3 相对误差: 9.06e-07	W3 相对误差: 9.56e-07	W3 相对误差: 4.96e-07
W4 相对误差: 3.09e-07	W4 相对误差: 2.76e-07	W4 相对误差: 7.76e-08
b1 相对误差: 7.17e-08	b1 相对误差: 4.12e-07	b1 相对误差: 3.55e-07
b2 相对误差: 1.53e-07	b2 相对误差: 2.33e-08	b2 相对误差: 3.92e-08
b3 相对误差: 1.47e-09	b3 相对误差: 1.49e-09	b3 相对误差: 3.20e-09
b4 相对误差: 1.27e-10	b4 相对误差: 1.57e-10	b4 相对误差: 4.34e-10

如果网络编码顺利，深层网络在较小数据集上很容易产生过拟合现象，运行下列代码块，可视化输出结果如图 3-15 所示。确保网络在 20 个训练周期内训练正确率接近 100%，如果测试不成功，可能需要重新仔细挑选权重取值范围（权重标准差）超参数 w 及学习率 l，祝你好运！

在小数据集上测试训练代码块：

```
input_size = 32 * 32 * 3
num_classes = 10
num_train = 50
X_train_small = X_train[ : num_train ]
y_train_small = y_train[ : num_train ]
w = 1e-1
l = 1e-3
net = FullyConnectedNet( input_size , [ 100, 100, 100, 100, 100 ] ,
                                        num_classes, weight_scale = w ,reg = 0.6 )
stats = net.train( X_train_small, y_train_small, X_val, y_val,
                    num_iters = 20, batch_size = 25,
                    learning_rate = l, learning_rate_decay = 0.95,
                    verbose = True )
# 绘制损失值历史曲线。
plt.subplot( 2, 1, 1)
plt.plot( stats[ 'loss_history' ], 'o' )
plt.title( 'Loss history' )
plt.xlabel( 'Iteration' )
plt.ylabel( 'Loss' )
# 绘制训练/验证精度历史曲线。
plt.subplot( 2, 1, 2 )
plt.plot( stats[ 'train_acc_history' ], label = 'train' )
plt.plot( stats[ 'val_acc_history' ], label = 'val' )
plt.title( 'Classification accuracy history' )
plt.xlabel( 'Epoch' )
plt.ylabel( 'Clasification accuracy' )
plt.show( )
```

如图 3-15 所示，由于网络能力过强，当使用训练数据集较少时，网络很快就产生了严重的过拟合现象，即使训练数据精度达到 100%，验证数据精度也只是随机“乱猜”情况下的 10%正确率。同时也有可能发现，深层神经网络是非常脆弱的，如果超参数选择不当，在大多数超参数配置下，网络都可能处于混沌状态，这使得训练深层神经网络变得十分困难。

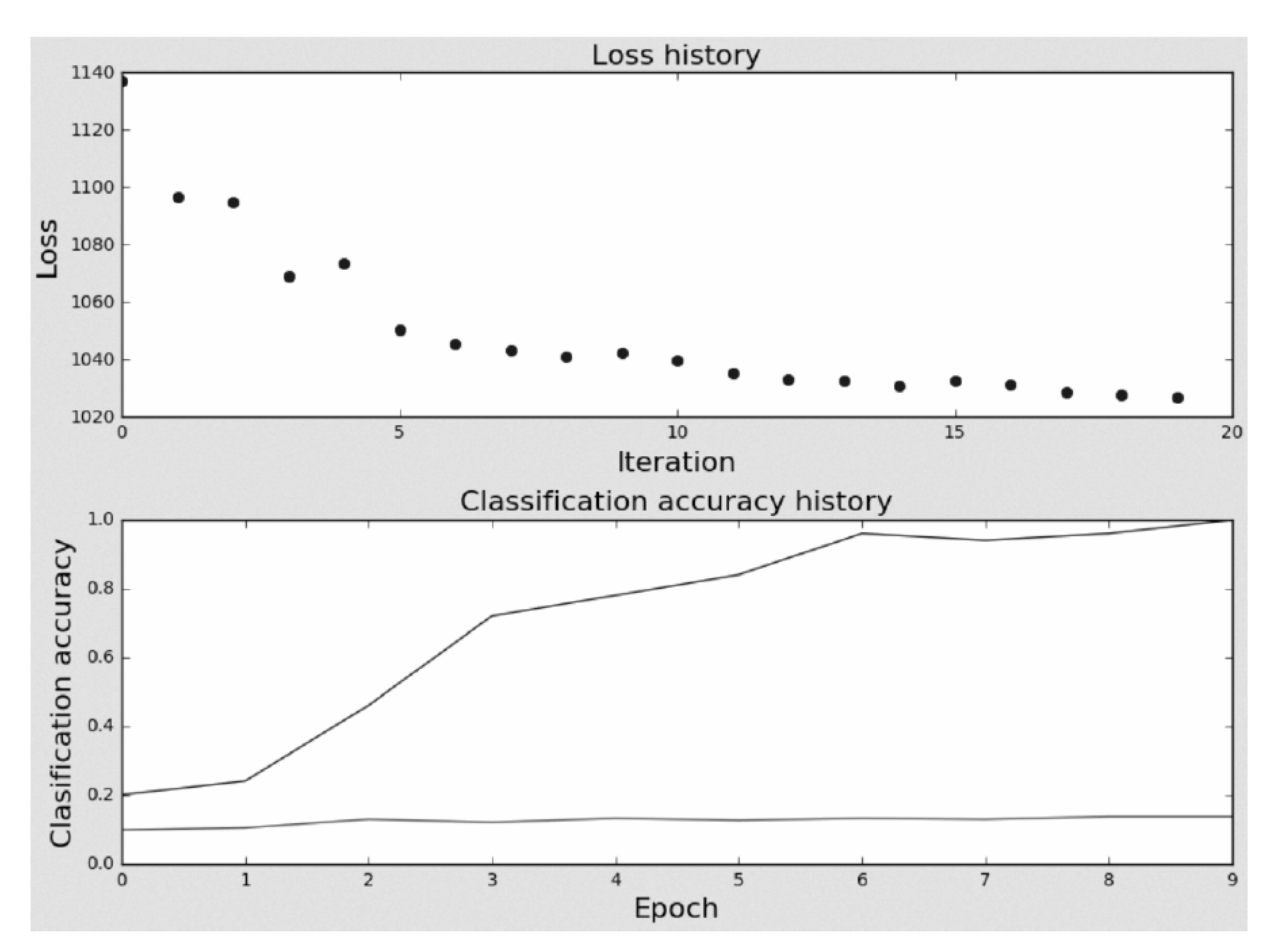


图 3-15　深层网络训练少量数据过拟合示意图

现在，我们使用完整的数据集测试网络，运行下列代码，深层网络训练示意图如图 3-16 所示。

深层网络性能测试代码块：

```
input_size = 32 * 32 * 3
num_classes = 10
net = FullyConnectedNet( input_size, [ 100, 100 ], num_classes , reg = 0.6, weight_scale = 2e-2 )
# 训练网络。
stats = net.train( X_train, y_train, X_val, y_val,
                num_iters = 2000, batch_size = 500,
                learning_rate = 8e-3, learning_rate_decay = 0.95,
                verbose = False )
# 测试性能。
val_acc = ( net.predict( X_val ) == y_val ).mean( )
print '验证精度: ', val_acc
print '最佳验证精度: ', stats[ 'best_val_acc' ]
# 绘制损失值历史曲线。
plt.subplot( 2, 1, 1 )
plt.plot( stats[ 'loss_history' ], 'o' )
plt.title( 'Loss history' )
plt.xlabel( 'Iteration' )
plt.ylabel( 'Loss' )
```

```
# 绘制训练/验证精度历史曲线。
plt.subplot( 2, 1, 2 )
plt.plot( stats[ 'train_acc_history' ], label = 'train' )
plt.plot( stats[ 'val_acc_history' ], label = 'val' )
plt.title( 'Classification accuracy history' )
plt.xlabel( 'Epoch' )
plt.ylabel( 'Clasification accuracy' )
plt.show( )
```

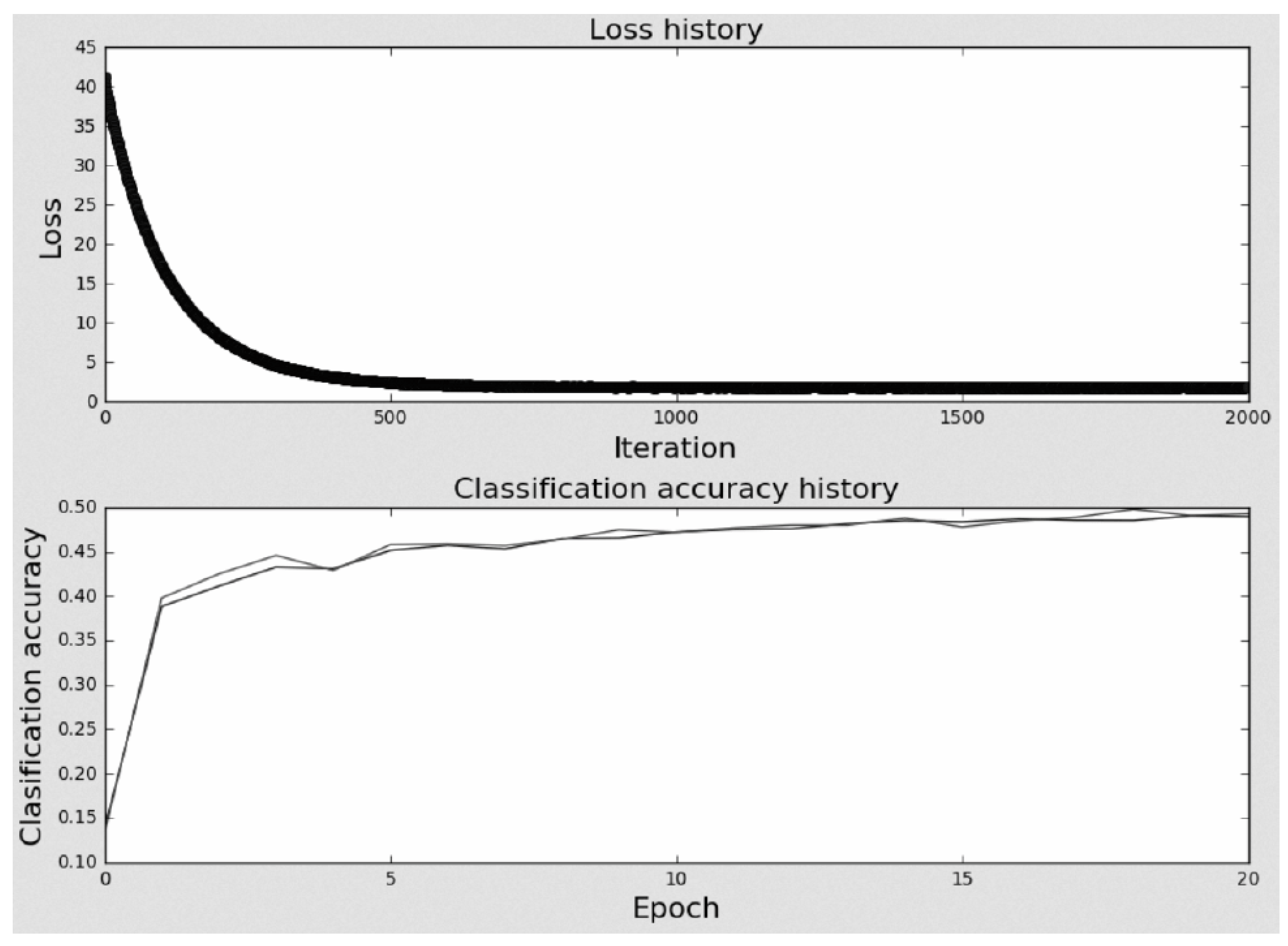


图 3-16　深层网络训练示意图

如图 3-16 所示，我们测试的深层网络的性能接近于 0.5，这连浅层网络都不如。接下来，你需要完成一个艰巨的任务，那就是训练出一个最佳的深层网络模型。虽然理论上深层网络的性能会比浅层网络好很多，但其训练难度十分大，能不能完成，就需要你的毅力了，加油！

3.5　参考代码

affine_forward 函数代码块：

```
def affine_forward( x, w, b ) :
    out = None
    N = x.shape[ 0 ]
    x_new = x.reshape( N, -1 )
    out = np.dot( x_new, w ) + b
```

```
    cache = ( x, w, b )
    return out, cache
```

affine_backward 函数代码块：

```
def affine_backward( dout, cache ) :
    x, w, b = cache
    dx, dw, db = None, None, None
    db = np.sum( dout, axis = 0 )
    xx = x.reshape( x.shape[ 0 ], -1 )
    dw = np.dot( xx.T, dout )
    dx = np.dot( dout, w.T )
    dx = np.reshape( dx, x.shape )
    return dx, dw, db
```

relu_forward 代码块：

```
def relu_forward( x ) :
    out = None
out = np.maximum( 0, x )
    cache = x
    return out, cache
```

relu_backward 代码块：

```
def relu_backward( dout, cache ) :
    dx, x = None, cache
    dx = dout
    dx[ x <= 0 ] = 0
    return dx
```

affine_relu_forward 函数代码块：

```
def affine_relu_forward( x, w, b ) :
    a, fc_cache = affine_forward( x, w, b )
    out, relu_cache = relu_forward( a )
    cache = ( fc_cache, relu_cache )
    return out, cache
```

affine_relu_backward 函数代码块：

```
def affine_relu_backward( dout, cache ) :
    fc_cache, relu_cache = cache
    da = relu_backward( dout, relu_cache )
```

```
    dx, dw, db = affine_backward( da, fc_cache )
    return dx, dw, db
```

shallow_layer_net.init 代码块：

```
def __init__( self, input_dim = 3 * 32 * 32, hidden_dim = 100, num_classes = 10,
              weight_scale = 1e-3, reg = 0.0 ) :
    self.params = { }
    self.reg = reg
    self.params[ 'W1' ] = weight_scale * np.random.randn( input_dim, hidden_dim )
    self.params[ 'b1' ] = np.zeros( hidden_dim )
    self.params[ 'W2' ] = weight_scale * np.random.randn( hidden_dim, num_classes )
    self.params[ 'b2' ] = np.zeros( num_classes )
```

shallow_layer_net.loss 代码块：

```
def loss( self, X, y = None ) :
    scores = None
    out1, cache1 = affine_relu_forward( X, self.params[ 'W1' ], self.params[ 'b1' ] )
    scores, cache2 = affine_forward( out1, self.params[ 'W2' ], self.params[ 'b2' ] )
    # 如果 y 为 None 直接返回得分
    if y is None:
        return scores
    loss, grads = 0, { }
    loss, dy = softmax_loss( scores, y )
    loss + = 0.5 * self.reg * ( np.sum( self.params[ 'W1' ] * self.params[ 'W1' ] )
                          + np.sum( self.params[ 'W2' ] * self.params[ 'W2' ] ) )
    dx2, dw2, grads[ 'b2' ] = affine_backward( dy, cache2 )
    grads[ 'W2' ] = dw2 + self.reg * self.params[ 'W2' ]
    dx, dw1, grads[ 'b1' ] = affine_relu_backward( dx2, cache1 )
    grads[ 'W1' ] = dw1 + self.reg * self.params[ 'W1' ]
    return loss,   grads
```

初始化深层网络权重代码块：

```
def __init__( self, input_dim = 3 * 32 * 32,   hidden_dims = [ 50 ,50 ],
             num_classes = 10, reg = 0.0, weight_scale = 1e-3 ) :
    for i in xrange( self.num_layers ) :
        self.params[ 'W' + str( i + 1 ) ] = weight_scale * np.random.randn(
                                            layers_dims[ i ], layers_dims[ i + 1 ] )
        self.params[ 'b' + str( i + 1 ) ] = np.zeros( ( 1, layers_dims[ i + 1 ] ) )
```

深层网络损失函数代码块：

```
def loss( self, X, y = None ) :
  scores = None
  cache_relu, outs, cache_out = { }, { }, { }
  outs[ 0 ] = X
  num_h = self.num_layers - 1
  for i in xrange( num_h ) :
      outs[ i + 1 ], cache_relu[ i + 1 ] = affine_relu_forward( outs[ i ],
                                           self.params[ 'W' + str( i + 1 ) ] ,
                                           self.params[ 'b' + str( i + 1 ) ] )
  scores, cache_out = affine_forward( outs[ num_h ],
  self.params[ 'W' + str( num_h + 1 ) ] ,
  self.params[ 'b' + str( num_h + 1 ) ] )
  loss, grads = 0.0, { }
  dout, daffine={ }, { }
  loss, dy = softmax_loss( scores, y )
  h = self.num_layers -1
  for i in xrange( self.num_layers ) :
        loss + = 0.5 * self.reg * ( np.sum(
            self.params[ 'W' + str( i + 1 ) ] * self.params[ 'W' + str( i + 1 ) ] ) )
    dout[ h ], grads[ 'W' + str( h + 1 ) ], grads[ 'b' + str( h + 1 ) ] = affine_backward(dy, cache_out )
    grads[ 'W' + str( h + 1 ) ] + = self.reg * self.params[ 'W' + str( h + 1 ) ]
    for i in xrange( h ):
        dout[ h – i – 1 ], grads[ 'W' + str( h – i ) ], grads[ 'b' + str( h – i ) ] =affine_relu_backward( dout[ h – i ],
                                                          cache_relu[ h – i ] )
        grads[ 'W' + str( h – i ) ] + = self.reg * self.params[ 'W' + str( h – i ) ]
  return loss, grads
```

深层神经网络 train 函数代码块：

```
def train( self, X, y, X_val,   y_val, learning_rate = 1e-3, learning_rate_decay = 0.95,
                 num_iters = 100, batch_size = 200,   verbose = False ) :
  for i,j in self.params.iteritems( ) :
       self.params[ i ] + = - learning_rate * grads[ i ]
```

深层网络 predict 函数代码块：

```
    def predict( self, X ) :
        y_pred = None
        outs = { }
        outs[ 0 ] = X
```

```
        num_h = self.num_layers - 1
        for i in xrange( num_h ) :
            outs[ i + 1 ] , _ = affine_relu_forward(outs[ i ], self.params[ 'W' + str( i + 1 ) ],
                                                    self.params[ 'b' + str( i + 1 ) ] )
        scores , _ = affine_forward( outs[ num_h ], self.params[ 'W' + str( num_h + 1 ) ],
                                     self.params[ 'b' + str( num_h + 1 ) ] )
        y_pred = np.argmax( scores, axis = 1 )
        return y_pred
```

3.6　参考文献

[1] Hüsken, M., & Stagge, P. (2003). Recurrent neural networks for time series classification. Neurocomputing, 50, 223-235.

[2] Bryson, A. E., & Ho, Y. C. (1975). Applied Optimal Control: Optimization, Estimation, and Control: distributed by Halsted Press.

[3] White, B. W., & Rosenblatt, F. (1963). Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms. American Journal of Psychology, 76(4), 705.

[4] Mackay, D. J. C. (1989). The Evidence Framework applied to Classification Networks. Neural Computation, 4(5), 720-736.

[5] Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Computer Science.

[6] Namin, A. H., Leboeuf, K., Muscedere, R., Wu, H., & Ahmadi, M. (2009). Efficient hardware implementation of the hyperbolic tangent sigmoid function. Paper presented at the IEEE International Symposium on Circuits and Systems.

[7] Nair, V., & Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. Paper presented at the International Conference on Machine Learning.

[8] Jarrett, K., Kavukcuoglu, K., Ranzato, M. A., & Lecun, Y. (2009). What is the Best Multi-Stage Architecture for Object Recognition? , 30(2), 2146-2153.

[9] Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. Paper presented at the Proc. ICML.

[10] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. 1026-1034.

[11] Goodfellow, I. J., Wardefarley, D., Mirza, M., Courville, A., & Bengio, Y. (2013). Maxout Networks. Computer Science, 1319-1327.

[12] Gerald, V. B., Fisher, L. D., Heagerty, P. J., & Thomas, L. (1981). Discrimination and Classification: Wiley.

[13] Lecun, Y., & Bengio, Y. (1998). Convolutional networks for images, speech, and time series: MIT Press.

[14] Dugas, C., Bengio, Y., Bélisle, F., Nadeau, C., & Garcia, R. (2001). Incorporating

second-order functional knowledge for better option pricing. Advances in neural information processing systems, 472-478.

[15] Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks. Paper presented at the International Conference on Artificial Intelligence and Statistics.

[16] Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. Neural Networks, 2(5), 359-366.

[17] Montúfar, G., Pascanu, R., Cho, K., & Bengio, Y. (2014). On the Number of Linear Regions of Deep Neural Networks. Advances in neural information processing systems, 39(1), 122-123.

[18] Goodfellow, I. J., Bulatov, Y., Ibarz, J., Arnoud, S., & Shet, V. (2013). Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks. Computer Science.

[19] Rumelhart, D. E., Mcclelland, J. L., & Group, T. P. (1986). Parallel Distributed Processing: Explorations in the Microstructures of Cognition. Language, 63(4).

[20] Hinton, G. E., & Anderson, J. A. (1981). Parallel Models of Associative Memory: Lawrence Erlbaum Associates.

[21] Cortes, C., & Vapnik, V. (1995). Support-vector networks. Machine Learning, 20(3), 273-297.

第 4 章 深度学习正则化

还记得机器学习的两个核心任务吗？首先，**尽可能地降低训练错误率**，上一章介绍的神经网络便是一种通用的函数逼近算法。只要给予足够多的神经元，理论上就可以逼近足够低的训练错误率，但也很容易出现机器学习的第二个核心问题，也就是严重的**过拟合现象**。那如何使训练错误率与测试错误率的差距尽可能的小呢？这便是本章讨论的主要内容，**正则化**（Regularization）。在机器学习中，所谓正则化，就是**降低验证错误率（有时需要牺牲训练错误率）的一系列方法**。

对于深度学习研究者而言，幸运的是我们拥有一个通用算法，只要我们愿意，把神经元增加、增加、再增加便可以得到非常低的训练错误率。但不幸的是，模型能力增大，其训练的难度也会剧增。即使将网络训练得非常优秀，也有可能面临严重的过拟合风险，使得“学习”变得毫无意义。因此我们需要不断地调整，限制设计的网络模型，从而降低过拟合风险。本章将介绍一些常用的深度学习正则化措施，该领域是深度学习非常重要的研究领域，从广义上来说，卷积神经网络和循环神经网络都可以算是从仿生学角度启发的正则化措施。在学习本章内容之前，希望你已经熟悉诸如泛化、欠拟合、过拟合等基础知识，如果不熟悉这些内容，那应该停下你匆忙的脚步，回到第 2 章中去仔细思考这些内容。

从降低泛化错误率的角度出发，伟大的前人们已经提出了大量的正则化策略，这其中最重要的方法就是**限制学习算法能力**。比如限制神经元的数量，限制参数数目（连接权重数目），又或者在目标函数中增加一些额外的惩罚项，这可以看作是一种软参数限制，比如权重衰减正则化和稀疏性正则化。如果细心选择，这些额外的限制与惩罚可以显著地降低测试错误率，

在 4.1 节中，我们介绍的**参数范数惩罚**便是最具代表性的通用参数惩罚措施。

有时这些限制与惩罚是我们对于某些任务的先验知识编码，比如图像识别需要旋转、平移不变性。因此我们使用卷积、池化、参数共享等限制措施去构建网络。在 4.2 节中，我们将介绍**参数绑定与参数共享**措施，在特定的任务中加入先验知识去限制网络。

在 4.3 节中，我们将介绍**噪声注入与数据扩充。**我们会进行“捣乱”，在输入数据或输出结果中**加入噪声**进行网络训练，就如同你看书时有人大吵大闹，你训练投篮时有人不停地对你进行干扰，这样训练出的网络便会有更强的健壮性，其泛化能力也就更好。

如果从数据的角度思考过拟合现象，那可能就是“多”与“少”的问题。所谓的“多”是指学习相同重复的内容太多，就是训练周期太长；所谓的“少”指的是学习内容的多样性太少，便是数据量太少。平移到深度学习中，我们会介绍两种策略解决过拟合现象，第一种你已经很熟悉了，那就是给予更多的学习资料（数据）。如果数据量有些“捉襟见肘”，没关系，在 4.3 节中我们还可以“伪造”数据进行**数据扩充来**帮助网络训练。针对训练周期太长的问题，我们在 4.5 节中使用称为**早停**（Early-Stopping）的技术去提前终止学习。

所谓“三个臭皮匠赛过诸葛亮”，那三个“书呆子”是否匹敌一个“万金油”呢？在 4.6 节中，借用**集成学习**（Ensemble Learning）的思路，最后我们将介绍一种既简单又有用的深度学习正则化方法——**Dropout**。

需要注意的是，深度学习算法是一个极其复杂的模型，而正则化做的大部分措施都可以解释为控制模型的复杂度。但控制模型复杂度，并不是简单地找到模型适合的大小，适合的参数数量。相反，在实际应用中我们发现，最佳模型是那些模型规模很大，但拥有恰当正则化的模型。接下来我们就正式开始我们的深度学习正则化之旅。

4.1 参数范数惩罚

机器学习中最常用的正则化措施是去**限制模型的能力**，而这其中最著名的方法便是 L0、L1 与 L2 范数惩罚。

假设我们要拟合一群二次函数分布的数据，但我们并不知道其真实的分布规律（知道也就不用学习了）。学习的本质其实就是不停地尝试，我们先从一次函数进行尝试，然后是五次函数，九次函数，三次函数，二次函数等，然后我们选出其中效果最好的一个，那便是最佳模型。

以上方法可能读者会有些失望，但未尝不是一个好方法。当然，为了显得“高端”些，我们需要将上面的内容粉饰一下。我们知道九次多项式包含了前八次多项式，那么为了节省力气，我们想要八次多项式时，只需要将九次项系数设置为 0 就可以了。有了这种想法，多项式函数的模型选择，其实就变成了高次多项式系数的限制。

从高次多项式函数一直到二次多项式函数的尝试，其实就是在限制参数的个数，如式（4.1）所示的九次多项式。

$$f(x) = w_1 x^9 + w_2 x^8 + \cdots + w_8 x^2 + w_9 x^1 + w_{10} x^0 \tag{4.1}$$

到最佳的式（4.2）所示的二次多项式。

$$f(x)=0x^9+0x^8+\cdots+0x^3+w_8x^2+w_9x^1+w_{10}x^0 \tag{4.2}$$

其实就是高次项的系数为零，用数学符号表示，便如式（4.3）所示。

$$\begin{aligned}&\min(J(w))\\&s.t.w_1=w_2=\cdots=0\end{aligned} \tag{4.3}$$

这样的思路非常好，但就是不方便实现。在实际应用中，需要标记所有函数对应的参数，对于深度学习这种拥有百万级参数规模的学习模型来说，那简直是不可想象的。

因此我们需要放松一下严苛的限制，我们不从高到底，顺序地限制参数，仅仅是控制参数的数目。如式（4.4）所示，我们将参数不等于 0 的个数控制在 c 以内来达到限制模型的目的，而这种方法就被称为 **L0 范数惩罚**。

$$\begin{aligned}&\min(J(w))\\&s.t.\sum_{i=1}^{m}I\{w_i\neq 0\}\leqslant c\end{aligned} \tag{4.4}$$

虽然我们放松了限制，但以上的表达式还是不太完美，对于实际应用还是不太友好。那我们不妨再放松一下限制，不要求参数的非零数量被限制在某个范围内，但要求参数数值的总和要小于某个数值，如式（4.5）所示，这种对参数数值总和的限制就被称为 **L1 范数惩罚**[1]**，也被称为参数稀疏性惩罚。**

$$\begin{aligned}&\min(J(w))\\&s.t.\sum_{i=1}^{m}|w_i|\leqslant c\end{aligned} \tag{4.5}$$

虽然高次项的系数很可能不为零，但如果发生如式（4.6）所示的情况，那高次项也可被轻松地忽略了。

$$f(x)=0.0001x^9+0.0003x^8+\cdots+0.002x^3+w_8x^2+w_9x^1+w_{10}x^0 \tag{4.6}$$

由此，我们的机器学习问题就变成了一个求条件极值的数学问题。但这还不完美，因为带有绝对值。我们再次放松限制，将对参数的绝对值求和改成对参数的平方求和，如式（4.7）所示，这就是 **L2 范数惩罚**[2]，也就我们非常熟悉的**权重衰减惩罚**。

$$\begin{aligned}&\min(J(w))\\&s.t.\sum_{i=1}^{m}(w_i)^2\leqslant c\end{aligned} \tag{4.7}$$

我们可以通过 c 的值来控制学习算法模型的能力，c 越大模型能力就越强，c 越小模型能力就越弱。如果大家基础较好，该条件极值可以使用**拉格朗日乘子法**来求解。当然，如果早已忘了也没关系，接下来我们就仔细研究下权重衰减正则化。

4.1.1　L2 参数正则化

代价函数的梯度如图 4-1 实线等高线椭圆所示，而梯度的最小值为其中心点，如果没有任何限制，学习的过程就是每次沿着梯度等高线垂直的方向寻找极值。而图中的虚线所示就是权重衰减项，虚线圆的半径便是权重衰减项中的常数 c。

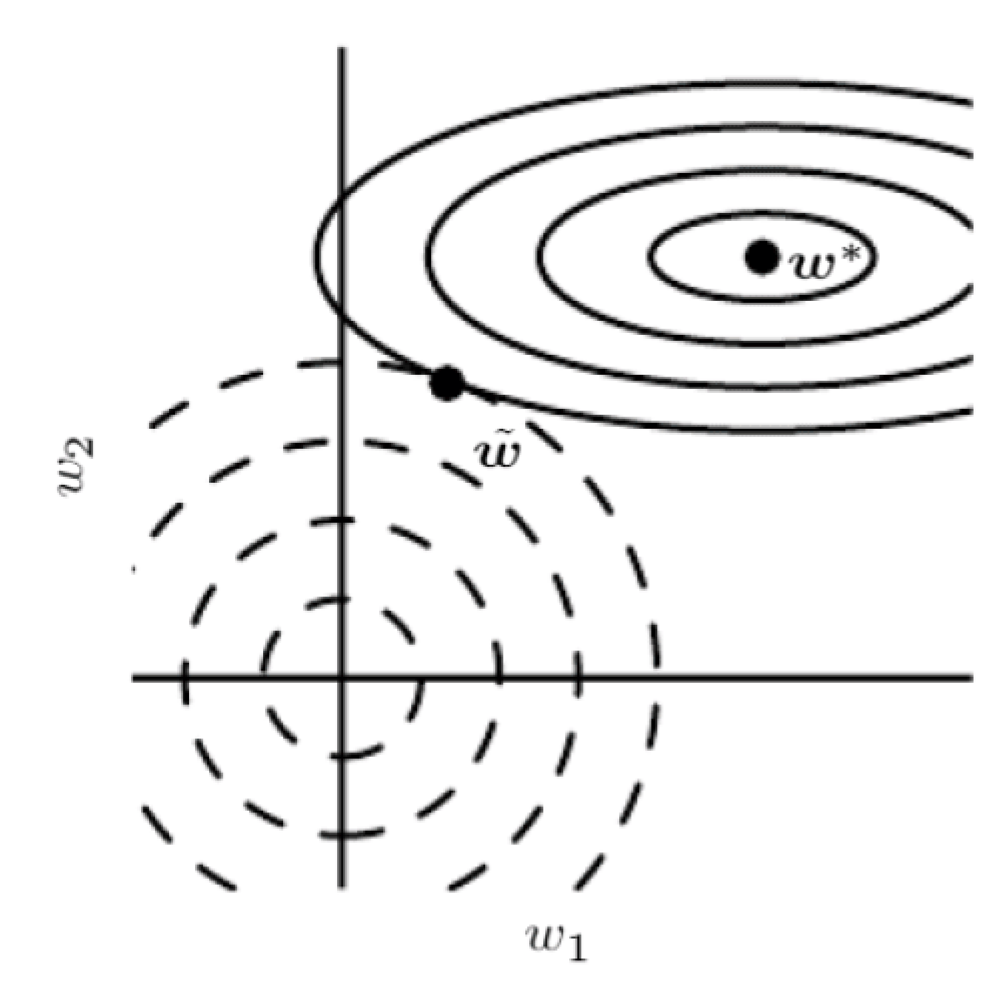


图 4-1　权重衰减几何意义示意图

这就如同一个带线的小球的一端被固定在原点，而我们拿着小球尽力地靠近椭圆的中心点。如果该小球的绳过长（c 值过大），小球几乎就没被限制，轻易地就到达了最小值，但也造成了过度拟合现象。如果绳子过短，那我们距离训练数据的最优值处就很远，也就造成了欠拟合现象。我们假设一种状态，绳长适中，那该状态下的最优效果是什么呢？

有兴趣的读者可以自己动手画一画，做一下试验，但其实这并不难想象，最近的点就在椭圆中心与原点连线上。此时又是一个什么状态呢？非常巧，此时代价函数的负梯度与固定一端的带绳小球的法向量是平行的，我们用数学符号刻画，便如式（4.8）所示。

$$\nabla(J(w))+2\lambda w=0 \tag{4.8}$$

需要说明的是带绳小球可以用向量 $\boldsymbol{w}^T\boldsymbol{w}<c$ 表示，因此该圆的法向量就为向量 $\boldsymbol{w}$，如果读者忘记了没关系，只需了解即可。其中常数 2 是我刻意添加的，只是为了构造函数方便，而 λ 为一个使得代价函数梯度与法向量平行的参数，而该参数便是大名鼎鼎的拉格朗日乘子，并且就如同绳长不能为负数，λ 也需要大于 0。

现在问题似乎变得有些复杂了，以前我们只需求解代价函数的梯度，然后通过梯度下降法来慢慢接近梯度为零的区域，但现在引入新的参数又如何解决呢？

在实际应用中，λ 通常作为超参数，人为进行选择，其值需要通过具体的验证数据测试结果进行经验性选择。现在我们把 λ 当作一个常数处理，但是代价函数的梯度依然无法求解。我们不妨这样思考，最小化代价函数等价于代价函数的梯度为 0。那代价函数梯度加上某个数为 0，那也等价于原代价函数加上某个函数。那我们不妨将其积分，如式（4.9）所示，这就是新的代价函数。

$$J_{新}(w) = J_{旧}(w) + \lambda \boldsymbol{w}^T \boldsymbol{w} \tag{4.9}$$

如果我们使用最小均方误差函数作为代价函数，那么该正则化代价函数就如式（4.10）所示。

$$J(w) = (y - f(w;x))^2 + \lambda \boldsymbol{w}^T \boldsymbol{w} \tag{4.10}$$

其中 λ 控制着机器学习算法模型的能力，如果 λ =0，则退化成原始模型，λ 值越大，则惩罚越严重，模型的能力越弱。λ 过小，容易发生过拟合现象，而 λ 过大，容易发生欠拟合现象。

在计算参数梯度时，如式（4.11）所示，也非常简单，其实就是在原代价函数梯度的基础上再加上 λ 倍的参数本身。我们这里把 2 省略了，因为 λ 和 2 都是常数，只需要调整 λ 即可。

$$\frac{\partial J_{新}(w)}{\partial w_i} = \frac{\partial J_{旧}(w)}{\partial w_i} + \lambda w_i \tag{4.11}$$

4.1.2　L1 正则化

权重衰减是最常用的正则化措施，其优点在于可导并且容易优化。但正如开始时介绍范数惩罚方法时，我们一次一次放松限制推出了权重衰减，如果想要更加严格地限制，L1 正则化便是一个不错的选择。如式（4.12）所示，便是加入 L1 惩罚的代价函数。

$$J_{新}(w) = J_{旧}(w) + \lambda \| w \|_1 \tag{4.12}$$

其中$\|w\|_1$ 表示所有参数的绝对值求和，并不是求向量模长。如果不太习惯，我们也可以将式（4.12）写成式（4.13）所示的表达式。

$$J_{新}(w) = J_{旧}(w) + \lambda \sum_{i=1}^{m} | w_i | \tag{4.13}$$

在计算参数梯度时，如式（4.14）所示，也非常简单，由于绝对值不可导，我们其实只是在原代价函数梯度的基础上再加上 λ 与参数本身的符号乘积。

$$\frac{\partial J_{新}(w)}{\partial w_i} = \frac{\partial J_{旧}(w)}{\partial w_i} + \lambda \mathrm{sign}(w_i) \tag{4.14}$$

其中 $\mathrm{sign}(x)$ 表示取 x 的符号，比如 $\mathrm{sign}(-5) = -1$，而 $\mathrm{sign}(5) = 1$。需要注意的是，相比于权重衰减，L1 正则化的限制更为严格，因此也就更加的稀疏（sparse），稀疏性也就是我们最终优化到的参数中有许多 0。稀疏性的一大好处是有利于**特征选择**（Feature Selection）[3]，由于大多数参数都为 0，而这些 0 参数对应的特征就没有被使用，实际上我们就选择出了一些重要特征对数据进行预测，并自动筛选掉一些无用的特征，这些无用特征很大程度上也有可能是噪声特征。

4.2 参数绑定与参数共享

目前为止，我们讨论的正则化措施都是通过限制模型参数的方法来限制模型的能力，但这种通用的限制太过宽泛，这里再次提起“没免费午餐理论”，机器学习算法没有好坏之分，所谓的好坏只是针对的数据集与问题域而言。迁移到正则化措施中，如果我们在特定领域对参数进行特定的限制，可能效果会更好。因此，我们经常需要添加一些先验知识去限制模型参数。虽然我们并不能精确地知道参数需要什么值，但对于特定的领域与特定的模型结构，模型参数之间存在着某些依赖。

假设某个同学数学非常有天赋，那他学习物理、化学和计算机等内容也会变得非常轻松。某个同学文学功底扎实，出口成章，那她在英语及艺术方面也可能高人一筹。在机器学习中，某个模型可以完美地识别“猫”，那通过训练，它也可以非常好地识别出“狗”。更具体些，识别“猫”使用的参数和识别“狗”使用的参数会比较接近。因此我们在训练其识别“狗”时，就可以利用识别“猫”的参数进行训练，这种重复利用参数在不同任务中的方法也被称为**多任务学习**（Multi-task Learning）[4]。

在图像识别领域，图像应该具有平移、旋转等空间不变性特征；而对于时间序列数据，数据与数据之间应该具有时间不变性特征。例如，一幅图中有一只小猫，如果将该图中的每个像素都向右平移一个像素点，这只小猫依然在图中。对于人而言，平移一个像素点根本分辨不出区别，但对于机器学习算法而言，所有的像素特征对应的参数都发生了偏移，而机器学习算法很可能就认为这是两幅图像。为了提取到这种不变性特征，我们迫使相邻数据特征参数相同，而这种正则化方法也称为**参数共享**（Parameter Sharing）。

卷积神经网络（Convolutional Neural Network，CNN）[5]是目前最流行的一种神经网络，广泛应用于计算机视觉领域。而该网络的核心——卷积操作，其实就是参数共享。参数共享使得卷积网络极大地降低了参数的规模，并且在不增加训练数据量的前提下，增加了网络的尺寸。我们将会在第 6 章中详细地介绍卷积网络的各项工作原理。

4.3 噪声注入与数据扩充

在机器学习中，想要降低泛化错误率最好的方法是**训练更多的数据**，但在现实中，数据总是有限的，并且还需要大量的成本。那么问题就来了，更多的数据使得模型性能更好，但我们又没有更多的数据，那该怎么办呢？其中一个有用而高效的方法就是生成**伪造数据**（fake data），然后将这些数据添加到训练集中进行**数据扩充**（Data Augmentation）[6]。虽然听上去似乎在教大家“造假”，但对于某些机器学习任务而言，创造新的伪造数据是非常合理的。

在分类任务中，分类器需要使用复杂高维的输入数据 x 以及数据所对应的类标 y 进行关联学习。这也意味着，分类器面临的主要任务是从各种变化的数据中获得某种稳定分布的不变性。既然平移、旋转后的图像都是同一个对象，那么在训练数据中扭曲一定程度的输入数据 x，分类器也应该将该数据与 y 进行关联，但这一方法的应用范围还比较窄。例如在密度估计任务中，要生成新的伪造数据是非常困难的，因为生成新数据的前提是我们已经解决了密度估计问题。

生成伪造数据最佳的应用场景是一类特定的分类任务——对象识别。图片是一种高维复

杂的数据，并且有着平移、旋转不变性等特点。在前一小节中，介绍了如果一幅图片全部向右平移一个像素，那对于机器学习算法来说，便彻底成了一幅新图像，因此我们需要先验地加入参数共享等措施来学习网络。那我们不妨换个角度，我们将这些先验知识加入到数据中去，即平移、旋转、拉伸之后的图像数据依然是原数据对象。我们将这些生成的“新图像数据”加入训练集中训练，那训练数据就可能增加十余倍的数据量，这样训练出来的模型同样也可以具备这种空间不变性能力，但我们必须谨慎地去变换这些数据。例如在字符识别中，将‘b’和‘d’水平翻转，‘6’和‘9’旋转 180°，那就都成了同一个数据。此时的伪造数据，就真的变成了错误数据。

比较机器学习基准结果时，需要注意数据扩充的影响。有时手工设计的数据扩充方法能够显著地降低泛化错误率，因此比较两种算法的性能时，执行控制试验就是非常有必要的。比较算法 A 与算法 B 时，我们需要确保这两种算法使用相同的手工设计数据扩充，假设算法 A 在没有数据扩充时表现糟糕，而算法 B 在加入人工合成数据后性能优越，这其实说明不了什么。在这样的例子中很有可能是合成数据造成了性能的提高，而不是算法 B 本身拥有更好的性能。但是否在试验中注入噪声或扩充数据是一种非常主观的判断，例如，在机器学习算法的输入中注入高斯噪声被认为是机器学习算法的一部分，然而对图片进行随机剪裁却被认为是单独地预处理步骤。

- 在输入中注入噪声

数据扩充也可以看作是在输入数据中注入噪声[7]，从而迫使算法拥有更强的健壮性（也可以将注入噪声看作是一种数据扩充）。神经网络容易过拟合的原因之一就是对于噪声没有太好的抗噪能力。正所谓“你怕什么，就应该勇敢地去面对什么。”最简单的提高网络抗噪声能力方法，就是在训练时加入随机噪声一起训练。比如在非监督学习算法**降噪自动编码器**（Denoising Autoencoder）[8]中，在输入层进行噪声注入然后学习无噪声的图片，便是一种很好的提高网络健壮性的方式。

- 在隐藏层注入噪声

对于某些模型，注入噪声也相当于对参数进行范数惩罚[9]。通常而言，注入噪声要比简单地收缩参数更有效，尤其是将噪声注入到隐藏层中。而隐藏层注入噪声是一个重要的主题，我们将在 4.6 小节 Dropout[10]算法中重点介绍，该方法也可以看作是通过加入噪声重构新输入的一种正则化策略。

除了在数据以及隐藏层输入中注入噪声，在权重（参数）中注入噪声也是一种有效的正则化措施。这种方法在某种程度上可以等价地解释为传统的范数惩罚，该方法鼓励参数找到一个参数空间，而该空间对于微小的参数变化所引起的输出变化的影响很小。换而言之，就是将模型送进了一个对于微小变化不敏感的区域，我们不仅找到了最小值，我们还找到了一个**宽扁的最小值区域**[11]。

- 在类标中注入噪声

数据集中或多或少都带有错误的类标 y，如果我们使用这些数据进行训练，那这就好比一个专心听讲且对老师十分尊敬的好学生，认真学习老师教给他的所有知识，但老师教给他的

却是错误的知识。最不幸的是，学生还对老师深信不疑，即使发现了错误，也只会认为是自己愚钝，不断地否定自己，不断地在痛苦中挣扎。虽然我们竭尽全力地避免错误标记数据，但这并不如我们想象得那么美好，人为地添加错误在所难免。与其小心翼翼地标记数据做个完美的“老师”，还不如直接就告诉学生“老师”并不完美，“老师”也会犯错。而解决这种问题的具体措施就是**在类标中加入一定的噪声**[12]，例如可以设置一个很小的常数ε，我们将训练数据的类标 y 正确的概率设置为 1-ε。

4.4 稀疏表征

在 4.1 参数范数惩罚小节中，我们通过**直接限制**可用参数规模，如 L1、L2 参数惩罚，来限制模型的能力，从而降低过拟合风险。接下来，我们介绍一种通过作用在神经网络激活单元的惩罚策略，来提高隐藏层的稀疏性，该方法可算作是一种**间接的**模型参数惩罚。

生物进化的一个重要趋势就是尽可能地节约能源，而我们的大脑也符合节能的原则。大脑中每个神经元都连接着成千上万的神经元，但其实大部分神经元都是处于抑制状态，只有少部分的神经元接受刺激会处于激活状态，因此我们需要将特定的信息交给特定的神经元进行处理。受此启发，我们就期望使用某种惩罚措施来抑制大部分神经元。当信息输入进神经网络时，只有关键部分神经元才能够处于激活状态，这就是**稀疏表征**（Sparse Representations）[13]。

我们已经讨论过 L1 范数如何致使参数稀疏化，这就意味着大多数参数都为零或接近于零，而表征稀疏化描述的其实就是隐藏层的输出大多数为零或接近零。为了简明地阐述这些区别我们以一个线性回归为例，如式（4.15）所示，是一个参数稀疏化的线性回归模型；而式（4.16）是使用稀疏表征的线性回归。

$$\underset{\boldsymbol{y}\in\mathbb{R}^m}{\begin{bmatrix}18\\5\\15\\-9\\-3\end{bmatrix}}=\underset{\boldsymbol{A}\in\mathbb{R}^{m\times n}}{\begin{bmatrix}4&0&0&-2&0&0\\0&0&-1&0&3&0\\0&5&0&0&0&0\\1&0&0&-1&0&-4\\1&0&0&0&-5&0\end{bmatrix}}\underset{\boldsymbol{x}\in\mathbb{R}^n}{\begin{bmatrix}2\\3\\-2\\-5\\1\\4\end{bmatrix}} \tag{4.15}$$

$$\underset{\boldsymbol{y}\in\mathbb{R}^m}{\begin{bmatrix}-14\\1\\1\\2\\23\end{bmatrix}}=\underset{\boldsymbol{B}\in\mathbb{R}^{m\times n}}{\begin{bmatrix}3&-1&2&-5&4&1\\4&2&-3&-1&1&3\\-1&5&4&2&-3&-2\\3&1&2&-3&0&-3\\-5&4&-2&2&-5&-1\end{bmatrix}}\underset{\boldsymbol{h}\in\mathbb{R}^n}{\begin{bmatrix}0\\2\\0\\0\\-3\\0\end{bmatrix}} \tag{4.16}$$

虽然表征正则化与参数正则化（一个使得输出稀疏，一个使得参数稀疏）有些区别，但幸运的是它们都使用相同的机理来实现。如式（4.17）所示，表征范数惩罚也是通过在代价函数中添加一项表征惩罚项 $\Omega(h)$ 来实现表征稀疏化。

$$J_{新}(w)=J_{旧}(w)+\alpha\Omega(h) \tag{4.17}$$

其中α控制着惩罚项的惩罚力度，如果$\alpha=0$，则相当于没有任何惩罚，α越大，惩罚力度越大。

就如同在参数中使用 L1 惩罚，通过限制参数的总数值来诱使参数稀疏化一样。在表征惩罚中我们也使用和 L1 惩罚相同的方式：$\Omega(h)=\|h\|=\sum_i|h_i|$来诱使表征稀疏化[14]。当然，L1 惩罚并不是唯一诱导稀疏表征的方法，其中 KL 散度（KL divergence）[15]，又称相对熵（Relative Entropy）也是一种十分常用的方法，有兴趣的读者可以自行查阅相关资料，这里就不再介绍。

4.5　早停

目前为止，我们针对过拟合现象所做的正则化措施总结起来有两点：一是限制模型的能力，就如同“孤已天下无敌（训练错误率很低），让你双手又如何（参数，表征惩罚）”。二是不断地加入噪声，给自己增加麻烦，就如同“放一碗水在头顶扎马步，穿着加沙的背心去奔跑，最终目的是使自己更强壮”。而在本小节中，我们介绍的这种正则化方法就非常简单了，那就是“当个懒汉早些停下来”。这似乎和我们的哲学意境不太相符，在我们的思想中，我们接受的传统熏陶总是这样：“弟子勤奋看书，师傅问弟子懂否？弟子挠头羞愧道不知。师傅轻捻长须道，再看。几日之后，再问弟子懂否？弟子颤惊道，徒儿依旧愚昧。师傅小叹，再看……”我们的哲学意境总是让我们在重复中自我醒悟，发觉渺茫的慧根。但本节介绍的深度学习正则化策略却恰恰相反，我们将学习如何适可而止，我们不允许算法重复地学习太多次。

基于梯度下降的深度学习算法都存在一个训练周期的问题，训练的次数越多，训练错误率就越低。由于我们真正目的是降低测试错误率，因此我们期望验证错误率也与训练错误率有相似的曲线。但期望总是事与愿违，如图 4-2 所示，真实环境中的验证错误率并没有随着训练次数的增多而减少，而是形成了一种“U 型”曲线，验证错误率先减小后上升。那很自然地，我们就期望能在验证错误率的最低点（或最低点附近）停止训练算法，这就是所谓的**早停**（Early Stopping）[16]。

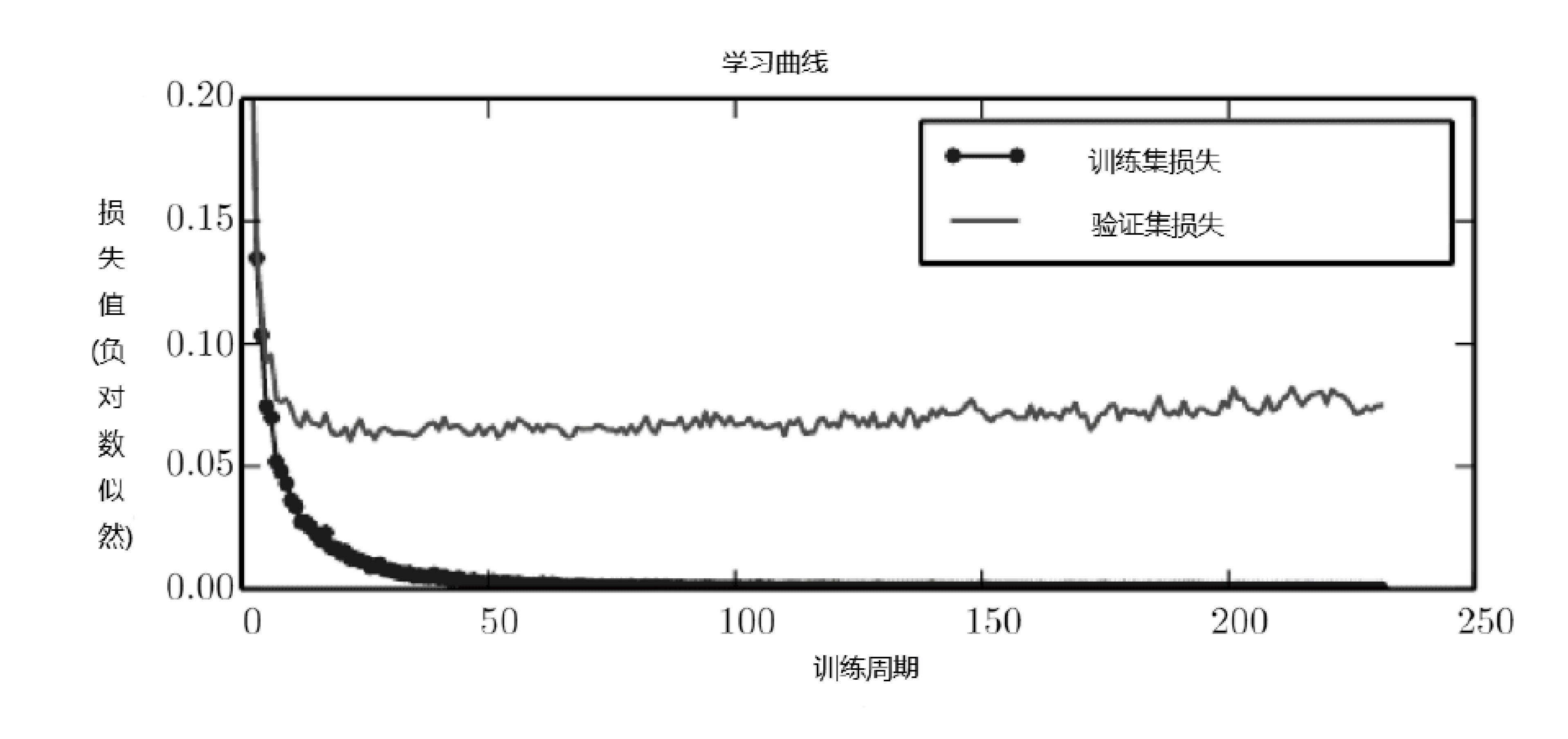


图 4-2　训练错误率、验证错误率与训练周期 U 型曲线

为了找到最佳的验证错误率，我们会保存一份最佳的模型参数，每经过一定的训练周期，我们就将当前的验证错误率与最佳错误率进行比较。如果当前验证错误率低于历史最佳错误

率，我们就将当前验证错误率设置为最佳验证错误率，然后将当前模型参数拷贝到最佳模型参数中。当训练次数足够多时，我们就返回最佳模型参数。如算法 4.1 所示，就详细地描述了这一过程。

算法 4.1 早停算法用于确定最佳模型参数以及最佳训练次数：

符号说明：

n 表示评估错误率的训练间隔数；p 表示“耐心值”，若进行 p 次训练后仍然没有获得更好的验证错误率，则中断训练；w_0 表示初始化参数。

训练步骤：

```
w = w0;   i = 0 ;   j = 0 ;   v = ∞ ;   w_temp = w ;   i_temp = i ;
While( j < p )
{
 训练 n 步算法后更新参数 w_tem;
 i_temp = i_temp + n ;
 V_temp = calValidationError( w ) ;
 if ( v_temp < v)
 {
   j = 0 ;   w = w_temp ;   i = i_temp ; v = v_temp ;
 } else {
   j = j + 1;
 }
}
Return w 最佳参数，i 最佳训练次数。
```

- 早停算法作为一种超参数

我们略显“玄幻”地解释了早停算法的好处，所谓“点到为止”似乎有些牵强。换个角度，可以将早停算法看作是一种高效的**超参数选择**。在这种观点中，训练次数可看作是一种超参数。不知道你是否还记得超参数的概念，如果已经模糊了，最好回到第 2 章再温习一下。选择超参数其实就是去寻找过拟合与欠拟合的最佳折中点，权重衰减中惩罚因子的取值，梯度下降中步长的选择都被称为超参数。同样地，我们通过图 4-3 可知，训练次数也存在过拟合与欠拟合的 U 型曲线，因此通过训练次数去限制模型的能力也就变成一种正则化措施了。

通常来说，超参数的选择是一个非常耗时、耗力的过程，但早停算法却非常高效。在单个训练过程中，我们仅仅拥有“训练次数”这一超参数，并且这种超参数是在训练过程中通过对验证错误率的比较自动选择出来的。我们需要付出的只是保存最佳参数所需的额外内存空间，但这种代价通常又是那么的微不足道。在实际训练中，最佳参数的更迭也并不频繁，因此对于总的训练时间所占的比重也就可以忽略了。

早停是一种非常“低调”的正则化措施[17]，它几乎不改变训练过程、目标函数或可选择的参数值，这也就意味着它在学习过程中几乎是无损失的。和权重衰减相比，不需要像权重衰减那样小心地设置惩罚因子，因为太大或太小的惩罚都会致使模型无法训练。早停算法既可单独使用也可以联合其他的正则化策略使用，并且也减少了没必要的训练过程，节约了有

效训练时间。

说得更简单些，早停算法其实就是用一堆验证数据作为监视器，当学习模型快要变成“书呆子”时给予“当头棒喝”，但这也引出一个不大不小的问题，我们需要划分出一部分训练数据作为监视器。也许读者并没有什么感触，觉得仅仅只是减少了点训练数据，并没有关系。但其实数据在机器学习中是一个关键性问题，回看近几年机器学习的大热，并不是因为发现了多么厉害的学习算法，根本原因有两个：一是**硬件技术更强大，训练速度更快**；二是大数据的来临，**我们拥有了令人羡慕的数据量**。当然，如果你是一个“土豪”，拥有足够多的数据，那划分一部分数据也没什么影响。但如果你是参加某项竞赛，又或是使用基准数据集测试算法性能，数据就会变得异常的宝贵，你比别人多利用一点数据也许你就可以摘得桂冠了。

为了最优地利用好数据，我们通常将早停的训练过程分为两个阶段：第一阶段如上文所描述的那样划分训练集与验证集，使用验证集监控训练，适当的时候就停止训练；第二阶段我们就利用所有的训练数据再次训练模型。第二阶段如何利用所有数据训练呢？再回看算法 4.1 的描述，应该注意到我们最终返回的是**最佳模型参数**以及**最佳训练次数**，那我们就围绕着这两个返回资源，提出两种基本策略作为参考。

- 再训练所有数据

第一种策略如算法 4.2 所示，是再次初始化模型，然后使用所有的训练数据再训练。在第一个训练阶段中，我们获取了最佳的训练次数，那在第二个阶段中我们就使用所有训练数据，训练和上一阶段最佳训练次数相同的训练量。但这种策略有些小瑕疵，例如，虽然在理论上数据越多，训练的泛化性能会越好，但我们并不能保证在第二轮训练中就会获得更好的泛化能力。

算法 4.2 使用早停算法确定最佳训练次数，然后使用所有训练数据再训练
1.X^{train} 和 y^{train} 分别表示训练数据以及训练数据标记；
2.将 X^{train} 和 y^{train} 划分成($X^{subtrain}$，X^{valid})和($y^{subtrain}$，y^{valid})；
3.使用($X^{subtrain}$，$y^{subtrain}$)作为训练数据，(X^{valid}，y^{valid})作为验证数据，运行算法 4.1 所示的早停算法，获取最佳训练次数 i；
4.再次随机初始化模型参数 w；
5.使用(X^{train}，y^{train})作为训练数据训练 i 次。

另一种策略，如算法 4.3 所示，就是利用最佳参数，然后在其基础上使用所有训练数据持续地训练。在该策略的第二阶段中，我们监控在验证数据集上的平均代价函数，我们会持续地训练直到错误率降低到一个满意的值后再中断训练。这种策略避免了从零开始再训练网络所产生的额外训练时间，但也存在着一些问题，由于我们并不能保证学习模型的错误率可以降低到期望的值，那我们也就无法保证该算法能够中断。

算法 4.3 使用早停算法确定过度拟合的开始区域
1.X^{train} 和 y^{train} 分别表示训练数据以及训练数据标记；
2.将 X^{train} 和 y^{train} 划分成($X^{subtrain}$，X^{valid})和($y^{subtrain}$，y^{valid})；
3.使用($X^{subtrain}$，$y^{subtrain}$)作为训练数据，(X^{valid}，y^{valid})作为验证数据，运行算法 4.1 所示的早停算法，获取当前最佳参数 w；

4.计算当前代价函数值 $\varepsilon = J(w, X^{subtrain}, y^{subtrain})$；
5. while$(J(w, X^{subtrain}, y^{subtrain}) > \varepsilon)$
{ 使用(X^{train}, y^{train})作为训练数据训练学习模型； }

4.6 Dropout

Dropout[10]是深度学习中一种计算廉价并且能力强大的正则化模型，其思想主要来源于集成学习中的 Bagging（Short for Bootstrap Aggregating）[18]方法。为了方便理解，我们接下来先简单地介绍下**集成学习**（Ensemble Learning）[19]。

4.6.1 个体与集成

集成学习通过构建并结合多个学习器来完成学习任务，换而言之，就是我们所谓的“集体主义”，通常先训练一组**个体学习器**（individual learner），然后再用某种策略将它们结合起来。个体学习器可以相同也可以不同，如果个体学习器相同，集成就称为是**同质的**（homogeneous），同质集成中的个体学习器就称为**基学习器**（base learner），如神经网络就是典型的神经元集成起来的；如果个体学习器不相同，那我们的集成就称为是**异质的**（heterogeneous），异质集成的个体学习器由不同的学习算法生成，而此时的个体学习器就不被称为基学习器，而称为**组件学习器**（component learner）。

其实同质集成与神经网络在最终结构上是一样的，之所以把神经网络和集成学习分开来看待是其组织方式的不同。集成学习是一种“自底向上”的构建方式，我们先要获得学习器，然后再关注如何将其组织起来形成一个整体。而神经网络通常是一种“自顶向下”的构建方式，我们先将各学习器构建起来，再关注如何学习调整结构内的每个学习器。

集成学习最重要的核心思想在于：集体比个体好。那集体凭什么就比个体好呢？要回答这个问题，我们需要再次谈谈机器学习的两个核心问题，欠拟合与过拟合问题。对应到集成学习领域也有两个主要的应对措施，那就是 Bagging 思想与 Boosting 思想。

欠拟合是因为模型能力不足引起的，再说得直观些就是我们的学习器只能学习到数据的一部分特征，这些特征不足以解释所有的数据特性。那如何解决呢？比如进行人脸识别任务，但我们训练的学习器的能力有限，只能识别局部特征，比如只能识别鼻子。那很简单，我们让接下来的学习器识别耳朵，再让下一个学习器识别眼睛，那这样集成出来的学习器就可以很好地识别人脸了。这种“弱弱联合”变强的方式在集成学习中就称为 Boosting[20]。

同样地，过拟合问题是因为模型能力太强所致，更直观些就是我们的学习器不仅能学习到数据的特征，还能学习到数据的噪声特征。这些特征足以解释所有的训练数据，但却严重影响了测试数据。就比如我们进行人脸识别，学习器不仅学习到眼睛、鼻子、耳朵等人脸特征，还学习到了训练数据中人的衣服、发型、甚至是雀斑等无用特征。虽然训练结果很好，但在测试数据中却不一定有这些特征，那测试结果必然糟糕。那我们如何使用集成方式去制约这种连“人脸雀斑”都能学习到的能力呢？那也很简单，比如第一个学习器除了学习到人脸特征外还能学习到“衣服特征”，下一个学习器还能学习除了人脸特征之外的“发型特征”，再下一个学习器还能学习“雀斑特征”。我们将这些学习器组合起来形成“强强制约”的情

况，这就是集成学习中 Bagging 的思想。

无论是“弱弱联合”的 Boosting，还是“强强制约”的 Bagging 都有一个核心，那就是学习器应该尽可能的多样。这里所说的“多样性”，不仅指集成学习器的数量越多越好，更重要的是学习器之间应该尽可能的“不同”，学习器的不同其实就是学习到数据特征的不同。如果学习器相似，那么获取到的特征也就相似，“弱弱联合”依然只是“臭皮匠”的愚蠢，而“强强组合”也只是加固其“书呆子”的偏执。

如何才能获取学习器的多样性呢？其实只有一个方式，那就是数据。从行为心理学的观点出发，所有人的思想行为都是一样的，人与人之所以看似不同，其实是接触到的生长环境不同造成的。当然，这种观点对于解释人而言肯定是不足的，但对于解释机器学习却很友好。如果我们使用相同的数据，相同的学习算法，那我们得到的学习器也就基本相同。对于同质集成，如果我们想要获取不同的学习器，那我们就应该给予其不同的数据进行训练。

对于 Bagging 而言，想要获取多样性的学习器，我们就通过从训练数据中随机采样一定的子数据集进行训练。训练得到学习器后，再将数据放回原始训练集中进行下一次随机采样来训练下一个学习器。如此重复进行，直到学习器的数目达到预设数量为止，这种方法也叫**自助采样法**（Bootstrap Sampling）。这种方法的关键点在于采样率的选择，如果采样率过高，采样数据集的相似性也就过高，学习器的“不同”就很难保证；如果采样率过低，那训练数据不足，个体学习器的性能就较差。在实践经验中，常采用 63.2%的采样率，但这并不是一条铁律，最佳的采样率还需要根据具体数据，具体问题进行实验。

对于 Boosting 而言，我们首先会使用所有训练数据去训练第一个学习器，但该学习器只会对某些数据识别较好，对于另一些数据识别较差（如果都好，就是强分类器了，不需要集成了）。接下来我们就使用下一个学习器，着重地学习前一个学习器识别**较差的数据**，而忽略之前识别较好的数据。如此重复进行，当学习器的数目达到预设数量后停止学习。

总之，依靠“好而不同”的个体，我们可以得到强而有力的集体。有了对集成学习最粗浅的认识后，接下来就开始介绍本节的主角——Dropout。

4.6.2　Dropout

如何将集成学习与神经网络结合起来呢？一个很自然的想法就是将神经网络当作基学习器（个体学习器）来训练，然后再将每个神经网络集成起来，这种方法最大的问题在于需要极大的训练时间及足够的内存消耗。神经网络最让人闻风丧胆的地方就是其可怕的训练时间，然后我们还想在此基础上训练多个神经网络，想想就可怕。当然，这确实也是一种好方法，如果条件允许，集成 5 到 10 个神经网络，其效果还是非常好的[12]。

有没有又可以集成多个神经网络，又可以节约训练时间的方法呢？在上文中，我们将神经网络当作基学习器来看待，而现在我们需要换一个角度，我们将神经网络当作最终集成起来的结果进行思考。但又不太一样，你可能很容易会想到，我们会将神经元当作个体学习器进行学习。但如果那样，我们其实又回到传统的同质集成循环中去了。如图 4-3（a）中的神经网络可以分解成 16 种不同的子网络，如图 4-3（b）所示，而 Dropout 所做的就是将神经网络中的弱子神经网络集成起来。

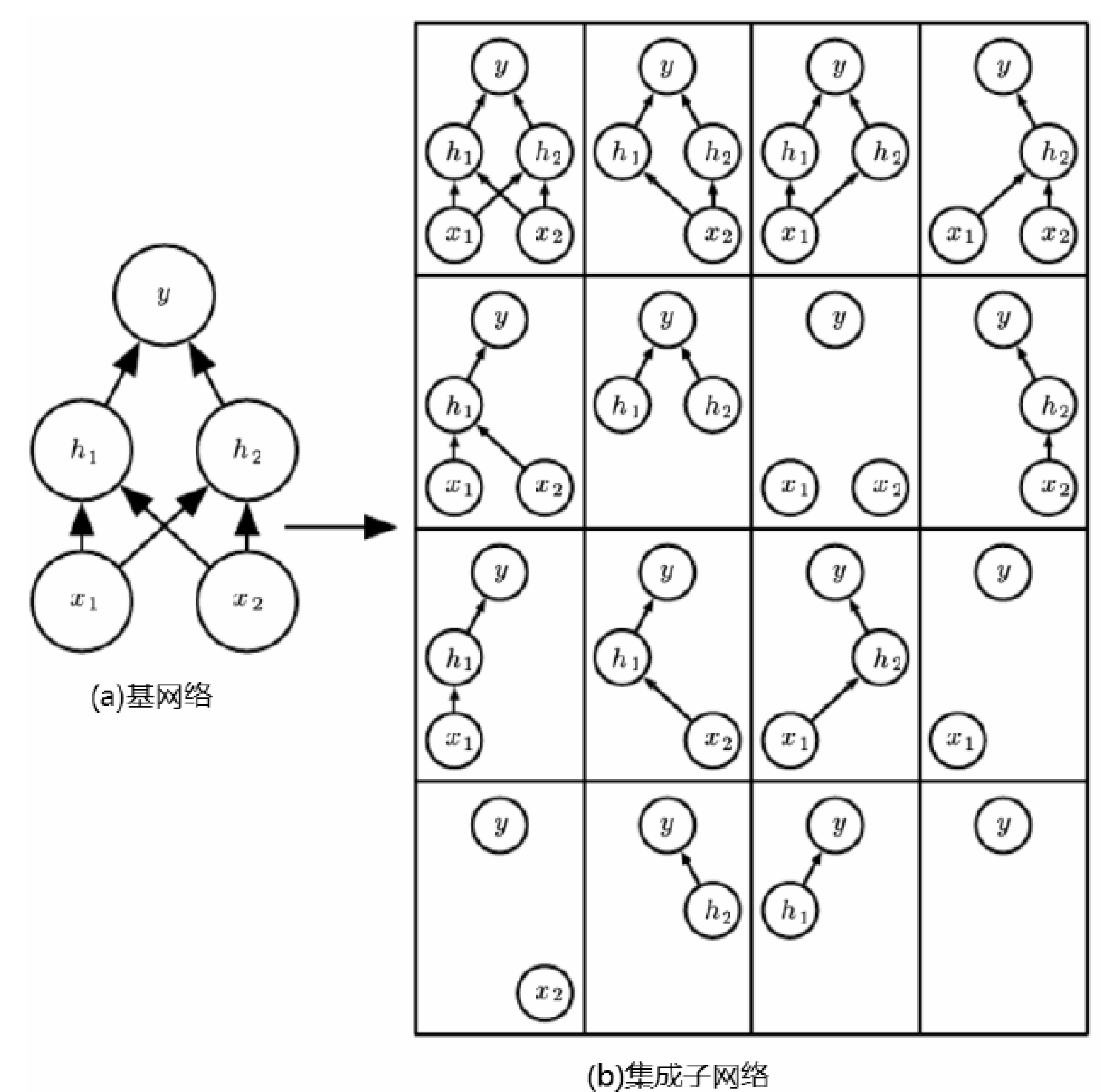
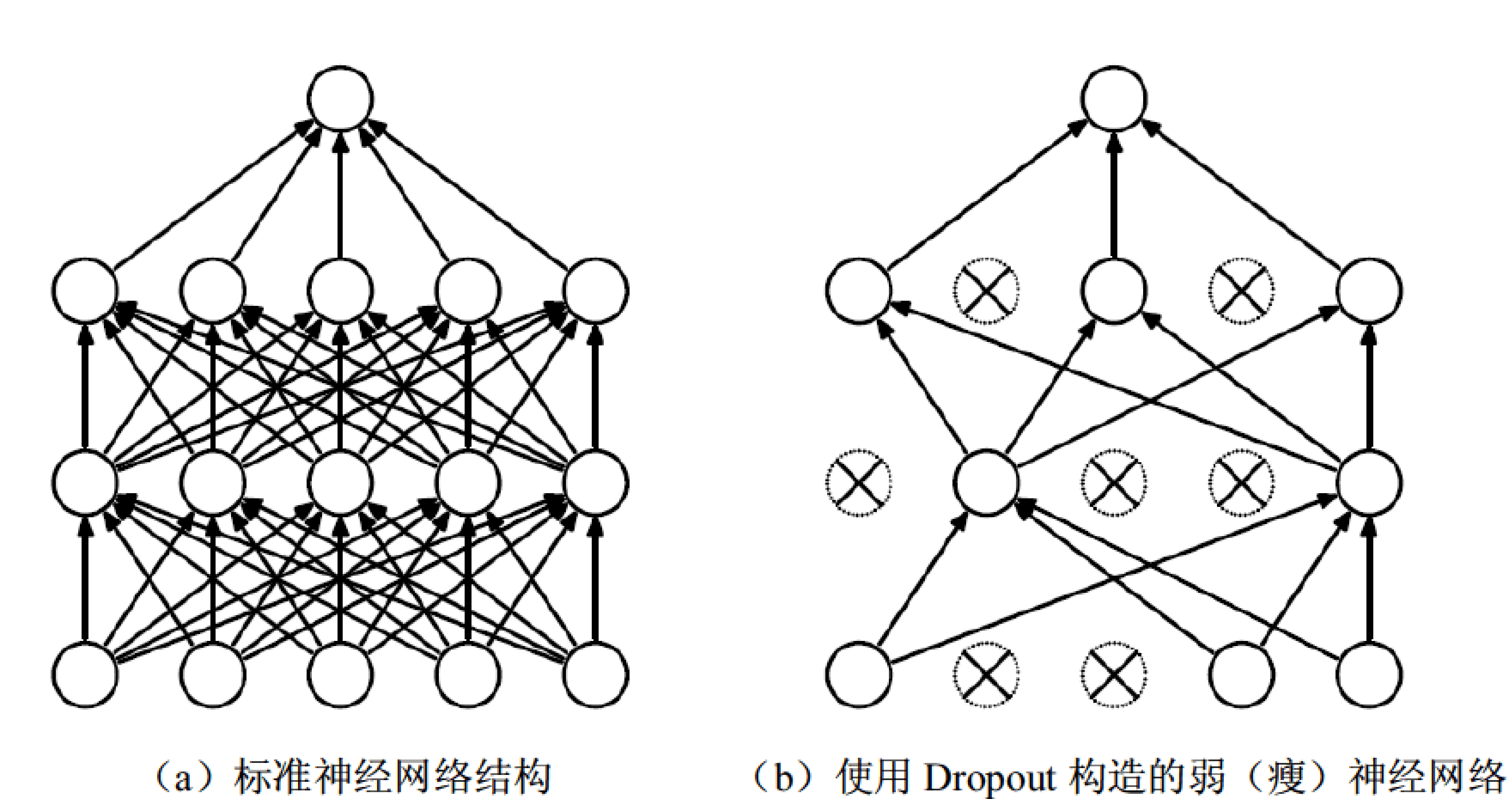


图 4-3　神经网络的结构分解示意图

在 Bagging 的思想中，我们想要获得不同的学习器，需要从训练数据中采样出不同的训练集。然后通过训练数据集的不同，从而期望获得不同“观点”的学习器。那么 Dropout 如何构造不同的弱神经网络进行训练呢？

其实非常简单，我们会在训练阶段给每个神经元独立地设置一个二项分布的“神经元激活”概率，若该值为 0，则表明当前神经元抑制；如果该值为 1，则表明当前神经元可用。如图 4-4（b）所示，为激活概率是 0.5 时，Dropout 所构造出弱神经网络的结构，该弱神经网络的神经元数量及参数数目（连接权重数量）都被极大地限制了，而图 4-4（a）为标准的神经网络结构。



（a）标准神经网络结构　　（b）使用 Dropout 构造的弱（瘦）神经网络

图 4-4　Dropout 神经网络模型

Dropout 分为训练阶段与测试阶段（执行阶段），训练阶段相比于传统的前馈神经网络，

只是多了一个**神经元采样**过程，而训练则采取随机梯度下降或最小批量梯度下降的方式进行。其实就是随机激活一定数量的神经元，然后执行一条或一小批数据，对激活的神经元进行误差反向传播修正权重，然后再次随机激活神经元训练网络。在测试或执行阶段，将神经元采样过程移除，退化为传统的神经网络进行测试。

Dropout 可以看作是集成学习中 Bagging 的一种形式，通过神经元采样构造不同的网络结构，然后通过采样出不同的数据来训练不同的弱神经网络。但这其中和 Bagging 还有一个非常不一样的地方，那就是**神经元共享**。Dropout 还有一个重要的灵感是来源于性别在生物进化中所承担的角色，为了更好地解释这一点，我们现在需要介绍一些生物学知识。

有性生殖是将父母中各自一半的基因，以及一些随机的变异混合在一起，然后产生后代。而无性生殖的后代是复制母体中全部的基因，然后掺入一点点变异。直观地看，似乎无性生殖对于优化个体的**适合度**（fitness）会更好些，因为母体已经组合好一些优良的基因，然后直接地复制给后代，后代就具有更良好的稳定性。另一方面，有性生殖破坏了原生物体基因间的相互适应性，降低了生物体已经进化出的环境适应能力。总而言之，无性生殖的后代变异更少，个体更稳定；有性生殖变异较大，个体不够稳定。然而，有性生殖却是大多数高级生物体的进化方式，所谓存在即合理，那么有性生殖又会有什么优越性呢？

原来在长期的生物进化中，自然选择的评价标准可能不只是个体的适合度，更需要考虑基因的混合能力。这种能力使得一组基因和另一组基因组合起来能够很好地工作，使基因间的组合更具健壮性。由于基因不能总是依赖父母遗传，其被迫从自身中学习获取优点，或者和其他的基因进行合作。根据这一理论，有性生殖的角色不仅仅是让新基因在种群中传播，而且也有利于降低基因间相互适应的复杂度。

同理，神经网络中的每个隐藏单元经过 Dropout 训练后，它也必须**学习与不同采样的神经元间的合作**，这使得神经元具有更强的健壮性，并且驱使神经元通过自身获取到有用特征，而不是依赖其他神经元去纠正自身的错误。在 Dropout 中，我们引入了一个神经元采样概率 p 超参数，在默认情况下该值设置为 0.5，但这不是一个定量，根据数据和网络的不同，还需要在试验中反复地测试。

虽然 Dropout 算法的执行过程非常简单，但却是一种非常高效的深度学习正则化措施，只需要在原始的网络中简单地添加一层神经元激活掩码就可能使网络性能得到大幅的提升。该方法是目前深度学习领域最常用的正则化措施之一，我们花费了大量的篇幅讲解该方法，希望你能在进入本章的编程练习前充分了解该方法的思想内涵。

纸上得来终觉浅，绝知此事要编程。接下来，我们就进入本章的编程练习。

4.7　深度学习编码实战中

本章的练习比较简单，权重衰减正则化的编码我们已经放在了第 3 章，而参数共享方法会在第 6 章卷积网络中重点描述，本节中我们主要完成 Dropout 正则化编码即可。随着学习的深入，我们在网络中添加的内容会越来越多，因此在本章节的练习中，我们需要将网络进行解耦，主要分为 trainer、updater 和 model 三个模块：trainer 负责配置训练过程中的一些超参数及可选项；updater 负责计算梯度，该部分将在第 5 章深度学习优化章节中具体介绍；model 为可选的神经网络，如 DNN、CNN 和 RNN 等。接下来，打开“第 4 章练习-神经网络中.ipynb”

文件，进入本章的练习。本章我们会逐步完成以下操作。

- 编码实现 Dropout 传播；
- 编码组合 Affine-ReLU-Dropout 层；
- 编码实现 Dropout 神经网络；
- 解耦神经网络；
- 正则化比较实验。

首先是我们已经非常熟悉的库文件导入代码，仅仅注意本章的练习文件存放在“DLAction/classifiers/chapter4”目录下即可。

库文件导入代码块：

```
# -*- coding: utf-8 -*-
import time
import numpy as np
import matplotlib.pyplot as plt
from classifiers.chapter4 import *
from utils import *
%matplotlib inline
plt.rcParams[ 'figure.figsize' ] = ( 10.0, 8.0 )
plt.rcParams[ 'image.interpolation' ] = 'nearest'
plt.rcParams[ 'image.cmap' ] = 'gray'
%load_ext autoreload
%autoreload 2
def rel_error( x, y ) :
  """ 返回相对误差 """
  return np.max( np.abs( x – y ) / ( np.maximum( 1e-8, np.abs( x ) + np.abs( y ) ) ) )
```

数据导入代码块：

```
# 载入预处理后的数据。
data = get_CIFAR10_data( )
for k, v in data.iteritems( ):
  print '%s: ' % k, v.shape
```

数据导入运行结果：

```
X_val:   (1000L, 3L, 32L, 32L)
X_train:   (49000L, 3L, 32L, 32L)
X_test:   (1000L, 3L, 32L, 32L)
y_val:   (1000L,)
y_train:   (49000L,)
y_test:   (1000L,)
```

4.7.1　Dropout 传播

由于 Dropout 的训练阶段和测试阶段采取不一样的传播方式，因此我们会设置“test”以及“train”模式。在训练模式时，我们会使用单独的神经元激活概率 *P* 生成 mask 掩码层。该掩码层中为“0”的位置表明该位置处神经元处于抑制状态，为“1”的位置表明该处神经元可用。在测试阶段，我们去除掩码操作，直接返回输入结果即可。

接下来打开“DLAction/ classifiers /chapter4/dropout_layers.py”文件，完成相应任务。

dropout_forward 函数代码块：

```
def dropout_forward( x, dropout_param ) :
  """
  执行 dropout 前向传播过程。
  Inputs:
  - x: 输入数据
  - dropout_param: 字典类型的 dropout 参数，使用下列键值：
    - p: dropout 激活参数，每个神经元的激活概率 p。
    - mode: 'test'或'train'，train：使用激活概率 p 与神经元进行"and"运算；
                            test：去除激活概率 p 仅仅返回输入值。
    - seed: 随机数生成种子。
  Outputs:
  - out: 和输入数据形状相同。
  - cache:元组( dropout_param, mask)：
        训练模式：掩码 mask 用于激活该层神经元，“1”为激活，“0”为抑制。
        测试模式：去除掩码操作。
  """
  p , mode = dropout_param[ 'p' ] , dropout_param[ 'mode' ]
  if 'seed' in dropout_param:
    np.random.seed( dropout_param[ 'seed' ] )
  mask = None
  out = None
  if mode == 'train':
    ############################################################
    #                任务：执行训练阶段 dropout 前向传播。        #
    ############################################################



    ############################################################
    #                          结束编码                          #
    ############################################################
```

```
elif mode == 'test':
    ##############################################################
    #              任务：执行测试阶段 dropout 前向传播。              #
    ##############################################################



    ##############################################################
    #                         结束编码                          #
    ##############################################################
cache = ( dropout_param,   mask )
out = out.astype( x.dtype, copy = False )
return out, cache
```

完成 dropout_forward 函数后，执行下列代码块进行检验。需要注意的是，Dropout 使一部分神经元失活，那么该层神经元的输出均值就会缩小 p 倍。例如我们输入均值为 10，$p = 0.3$，那输出的均值就会变为 3。这在零均值时没有影响，但如果我们的数据不是零均值分布，那可能会受到影响。为了消除这种影响，我们使用 mask=mask/p，这样就可以保证输出均值和输入均值相同，并且也间接放大了可用神经元的"影响力"。而现在当神经元处于激活状态时，其输出值会被放大 $1/p$ 倍。

测试 dropout_forward 函数代码块：

```
from classifiers.chapter4.dropout_layers import *
x = np.random.randn( 500, 500 ) + 10
for p in [ 0.3, 0.6, 0.75 ] :
    out, _ = dropout_forward( x, { 'mode': 'train', 'p': p } )
    out_test, _ = dropout_forward( x, { 'mode': 'test', 'p': p } )
    print '测试概率  p = ', p
    print '均值输入: ', x.mean( )
    print '训练阶段输出均值: ', out.mean( )
    print '测试阶段输出均值: ', out_test.mean( )
    print '训练阶段输出为 0 的平均个数: ', (out == 0).mean( )
    print '测试阶段输出为 0 的平均个数: ', (out_test == 0).mean( )
```

dropout_forward 函数编码正确后可能的测试结果：

测试概率 p = 0.3 均值输入: 10.0018991394 训练阶段输出均值: 9.99269850479 测试阶段输出均值: 10.0018991394	测试概率 p = 0.6 均值输入: 10.0018991394 训练阶段输出均值: 10.0121231455 测试阶段输出均值: 10.0018991394	测试概率 p = 0.75 均值输入: 10.0018991394 训练阶段输出均值: 9.99909975905 测试阶段输出均值: 10.0018991394

训练阶段输出为 0 的平均个数: 0.70024 测试阶段输出为 0 的平均个数: 0.0	训练阶段输出为 0 的平均个数: 0.39934 测试阶段输出为 0 的平均个数: 0.0	训练阶段输出为 0 的平均个数: 0.25026 测试阶段输出为 0 的平均个数: 0.0

- Dropout 反向传播

编码完成 Dropout 的前向传播后，接下来我们实现 Dropout 的反向传播。该过程同样会分为测试和训练两种模式。在测试阶段，仅仅返回上层梯度即可。在训练阶段，需要将处于抑制状态神经元所对应的上层梯度设置为 0。因此需要用到前向传播中的 mask，其已经放在 cache 中了，直接取出使用即可。

dropout_backward 函数代码块：

```
def dropout_backward( dout, cache ):
  """
  dropout 反向传播过程。
  Inputs:
  - dout:  上层梯度，形状和其输入相同。
  - cache:  前向传播中的缓存( dropout_param, mask )。
  """
  dropout_param, mask = cache
  mode = dropout_param[ 'mode' ]
  dx = None
  if mode == 'train' :
    ############################################################
    #                    任务：实现 dropout 反向传播。            #
    ############################################################


    ############################################################
    #                          结束编码                          #
    ############################################################
  elif mode == 'test' :
    dx = dout
  return dx
```

完成 Dropout 反向传播编码后，使用下列代码块进行梯度检验，相对误差应该小于 1e-10。

Dropout 反向传播梯度检验代码块：

```
from utils import *
x = np.random.randn( 10, 10 ) + 10
dout = np.random.randn( * x.shape )
```

```
dropout_param = { 'mode': 'train', 'p': 0.8, 'seed': 123 }
out, cache = dropout_forward( x, dropout_param )
dx = dropout_backward( dout, cache )
dx_num = eval_numerical_gradient_array( lambda xx: dropout_forward( xx, dropout_param )[ 0 ], x, dout )
print 'dx 相对误差: ', rel_error( dx, dx_num )
```

正确编码后可能的检验结果：

```
dx 相对误差:  5.44560766472e-11
```

4.7.2 组合 Dropout 传播层

如果在网络的隐藏层中使用了 Dropout 方法，那么一层神经网络就可以分解成三个处理阶段：affine 传播，ReLU 传播及 Dropout 传播。为了后续编码便捷，我们现在把这三个阶段整合在一起。接下来打开“DLAction/classifiers /chapter4/dropout_layers.py”文件，完成完整的 Dropout 前向传播，需要将各阶段的缓存保存在 cache 中。当然你也可以跳过该节的编码，但在进行下一小节的 Dropout 神经网络编码时，你需要在该网络中补全这三个阶段的处理过程。

affine_relu_dropout_forward 函数代码块：

```
def affine_relu_dropout_forward( x, w, b, dropout_param ):
  """
  组合 affine_relu_dropout 前向传播过程。
  Inputs:
  - x: 输入数据，其形状为(N, d_1, ..., d_k)的 numpy 数组。
  - w: 权重矩阵，其形状为( D, M )的 numpy 数组，D 表示输入数据维度，M 表示输出数据维度。
       可以将 D 看成输入的神经元个数，M 看成输出神经元个数。
  - b: 偏置向量，其形状为( M, )的 numpy 数组。
  - dropout_param: 字典类型的 dropout 参数，使用下列键值:
    - p: dropout 激活参数，每个神经元的激活概率 p。
    - mode: 'test'或'train'，train：使用激活概率 p 与神经元进行"and"运算；
                            test：去除激活概率 p 仅仅返回输入值。
    - seed: 随机数生成种子。
  Outputs:
  - out: 和输入数据形状相同。
  - cache:缓存包含( cache_affine, cache_relu, cache_dropout )，
          cache_affine：仿射前向传播的各项缓存；
          cache_relu：ReLU 前向传播的各项缓存；
          cache_dropout：dropout 前向传播的各项缓存。
  """
  out_dropout = None
  cache = None
```

```
    ################################################################
    #          任务：实现 affine_relu_dropout  神经元前向传播。          #
    #        注意：需要调用 affine_forward 及 relu_forward 函数，        #
    #           并将各自的缓存保存在 cache 中。                        #
    ################################################################



    ################################################################
    #                           结束编码                              #
    ################################################################
    return out_dropout, cache
```

完成 Dropout 神经元前向传播后，接下来我们完成 Dropout 神经元的反向传播，打开“DLAction/ classifiers /chapter4/dropout_layers.py”文件的 affine_relu_dropout_backward 函数，按要求完成相应的编码任务。

affine_relu_dropout_backward 函数代码块：

```
def affine_relu_dropout_backward ( dout, cache ) :
    """
     affine_relu_dropout 神经元的反向传播过程。
    Input:
    - dout: 形状为( N, M )的上层梯度。
    - cache: 缓存( cache_affine, cache_relu, cache_dropout )。
    Returns:
    - dx: 输入数据 x 的梯度，其形状为( N, d1, ..., d_k )。
    - dw: 权重矩阵 w 的梯度，其形状为( D, M )。
    - db: 偏置项 b 的梯度，其形状为( M, )。
    """
    cache_affine, cache_relu, cache_dropout = cache
    dx, dw, db = None, None, None
    ############################################################
    #          任务：实现 affine_relu_dropout 反向传播。           #
    ############################################################



    ############################################################
    #                        结束编码                           #
    ############################################################
    return dx, dw, db
```

4.7.3 Dropout 神经网络

接下来，我们将 Dropout 功能添加到第 3 章中的深层全连接神经网络中，打开“DLAction/classifiers/chapter4/fc_net.py”文件，完成相应的编码任务。

- Dropout 神经网络初始化

首先是比较熟悉的权重初始化环节，我们仅仅添加了 Dropout 参数，该参数为 0 时，表示不使用 Dropout 功能，网络将退回原来的全连接网络；若 Dropout 不为 0，则将使用该值作为激活概率进行 Dropout 传播。阅读下列代码块，对 Dropout 神经网络初始化，确保你熟悉且明白这些内容。

Dropout 神经网络初始化代码块：

```
def __init__(self, input_dim = 3 * 32 * 32, hidden_dims = [ 100,100 ],
num_classes = 10, dropout = 0, reg = 0.0, weight_scale = 1e-2, seed = None):
    """
    初始化全连接网络。
    Inputs:
    - input_dim: 输入维度。
    - hidden_dims: 隐藏层各层维度，如[ 100, 100 ]。
    - num_classes: 分类数量。
    - dropout: 如果 dropout = 0，表示不使用 dropout。
    - reg:正则化衰减因子。
    - weight_scale:权重范围，给予初始化权重的标准差。
    - seed: 使用 seed 产生相同的随机数。
    """
    self.use_dropout = dropout > 0
    self.reg = reg
    self.num_layers = 1 + len( hidden_dims )
    self.params = { }
    layers_dims = [ input_dim ] + hidden_dims + [ num_classes ]
    for i in xrange( self.num_layers ):
        self.params[ 'W' + str( i + 1 ) ] = weight_scale * np.random.randn(
                                        layers_dims[ i ], layers_dims[ i + 1 ] )
        self.params[ 'b' + str( i + 1 ) ] = np.zeros( ( 1, layers_dims[ i + 1 ] ) )
    self.dropout_param = { }
    if self.use_dropout :
      self.dropout_param = { 'mode' : 'train', 'p' : dropout }
      if seed is not None:
        self.dropout_param[ 'seed' ] = seed
```

- Dropout 神经网络损失函数

Dropout 神经网络的损失函数也与之前的全连接网络十分类似，由于我们之前已经编写了 affine_relu_dropout_forward 函数，该处代码应该很轻松即可完成。

需要注意的是，我们的网络目前拥有两种模式，Dropout 传播及无 Dropout 传播，因此可能需要编写如下的条件语句。

```
if self.use_dropout :
    Dropout 传播
else:
    正常传播
```

别忘了在输出层我们还需要使用仿射传播。祝你好运！

Loss 函数代码块：

```
    def loss( self, X, y = None ) :
mode = 'test' if y is None else 'train'
        # 设置执行模式。
        if self.dropout_param is not None:
            self.dropout_param [ 'mode' ] = mode
        scores = None
        ############################################################################
        #                   任务：执行全连接网络的前馈过程。                        #
        #               计算数据的分类得分，将结果保存在 scores 中。                #
        #   当使用 dropout 时，需要使用 self.dropout_param 进行 dropout 前馈。      #
        #       例如  if self.use_dropout: dropout 传播     else：正常传播          #
        ############################################################################



        ############################################################################
        #                               结束编码                                   #
        ############################################################################
        if mode == 'test':
            return scores
        loss, grads = 0.0, { }
        ############################################################################
        #        任务：实现全连接网络的反向传播。                          #
        #   将损失值存储在 loss 中，梯度值存储在 grads 字典中，            #
        #   注意网络需要设置两种模式：有 dropout 和无 dropout，            #
        #   例如 if self.use_dropout: dropout 传播，else：正常传播。       #
```

```
################################################################




################################################################
#                         结束编码                              #
################################################################
return loss, grads
```

- 梯度检验

完成上述任务后，我们使用下列代码块进行梯度检验。

Dropout 神经网络梯度检验代码块：

```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn( N, D )
y = np.random.randint( C, size = ( N, ) )
for dropout in [ 0, 0.2, 0.5, 0.7 ] :
  print '检验  dropout 率  =', dropout
  model = FullyConnectedNet( input_dim = D, hidden_dims = [ H1, H2 ],
                             num_classes = C, weight_scale = 5e-2, dropout = dropout, seed = 13 )
  loss, grads = model.loss( X, y )
  print '初始化  loss: ', loss
  for name in sorted( grads ) :
    f = lambda _: model.loss( X, y ) [ 0 ]
grad_num = eval_numerical_gradient( f, model.params[ name ],verbose = False, h = 1e-5 )
print '%s  相对误差: %.2e' % (name, rel_error( grad_num, grads[ name ] ) )
```

正确编码 Dropout 神经网络后可能的梯度检验结果：

检验 dropout 率 = 0	检验 dropout 率 = 0.2	检验 dropout 率 = 0.5
初始化 loss: 2.30679849759	初始化 loss: 2.29542906388	初始化 loss: 2.30729499497
W1 相对误差: 4.79e-06	W1 相对误差: 3.04e-05	W1 相对误差: 1.16e-07
W2 相对误差: 1.30e-07	W2 相对误差: 2.67e-09	W2 相对误差: 1.40e-06
W3 相对误差: 6.44e-08	W3 相对误差: 6.48e-09	W3 相对误差: 9.09e-09
b1 相对误差: 1.20e-08	b1 相对误差: 2.56e-08	b1 相对误差: 2.29e-09
b2 相对误差: 1.65e-09	b2 相对误差: 1.10e-10	b2 相对误差: 2.56e-09
b3 相对误差: 1.82e-10	b3 相对误差: 6.30e-11	b3 相对误差: 7.91e-11

4.7.4 解耦训练器 trainer

在本书后面的章节中，我们还将学习到很多不同的神经网络优化策略以及神经网络模型，

为了今后的编程方便，我们需要将模块进行解耦。首先，我们解耦模型的训练过程，将数据采样、衰减学习率、设置训练周期、是否打印中间结果等内容封装到 trainer 类中。接下来打开“DLAction/classifiers /chapter4/trainer.py”文件，阅读相关内容，确保自己熟悉整个过程。

- 训练器初始化

trainer 类在初始化时需要接受三个参数：训练模型、数据及一些可选项配置。可选项参数包括更新规则、学习率衰减系数、批量数据大小与训练周期等，更详细的内容请参考下列代码。

trainer 初始化代码块：

```
def __init__( self, model, data, **kwargs ) :
    """
    初始化训练器各项配置。
    必选参数：
    - model: 神经网络模型，如：DNN, CNN, RNN 等。
    - data: 数据字典，其中：
      'X_train': 形状为( N_train,   d_1, ...,   d_k )的训练数据。
      'X_val': 形状为( N_val,   d_1, ...,   d_k )的验证数据。
      'y_train': 形状为( N_train, )的训练数据类标。
      'y_val': 形状为( N_val, )的验证数据类标 。
    可选参数：
    - update_rule:更新规则，其存放在 updater.py 文件中，默认选项为'sgd'。
    - updater_config:更新规则所对应的超参数配置，同见 updater.py 文件。
    - lr_decay: 学习率衰减系数。
    - batch_size: 批量数据大小。
    - num_epochs: 训练周期。
    - print_every: 整数型，每迭代训练 print_every 次模型，打印一次中间结果。
    - verbose: 布尔型；是否在训练期间打印中间结果。
    """
    self.model = model
    self.X_train = data[ 'X_train' ]
    self.y_train = data[ 'y_train' ]
    self.X_val = data[ 'X_val' ]
    self.y_val = data[ 'y_val' ]
    # 弹出可选参数，进行相关配置。
    self.update_rule = kwargs.pop( 'update_rule',   'sgd' )
    self.updater_config = kwargs.pop( 'updater_config',   { } )
    self.lr_decay = kwargs.pop( 'lr_decay',   1.0 )
    self.batch_size = kwargs.pop( 'batch_size',   100 )
    self.num_epochs = kwargs.pop( 'num_epochs', 10 )
```

```
        self.print_every = kwargs.pop( 'print_every', 10 )
        self.verbose = kwargs.pop( 'verbose', True )
        # 若可选参数错误，抛出异常。
        if len( kwargs ) > 0:
            extra = ', '.join( '"%s"' % k for k in kwargs.keys( ) )
            raise ValueError( 'Unrecognized arguments %s' % extra )
        # 确认 updater 中含有更新规则。
        if not hasattr( updater, self.update_rule ):
            raise ValueError( 'Invalid update_rule "%s"' % self.update_rule )
        self.update_rule = getattr( updater,    self.update_rule )
        # 初始化相关变量。
        self.epoch = 0
        self.best_val_acc = 0
        self.best_params = { }
        self.loss_history = [ ]
        self.train_acc_history = [ ]
        self.val_acc_history = [ ]
        # 对 updater_config 中的参数进行深拷贝。
        self.updater_configs = { }
        for p in self.model.params :
            d = { k: v for k, v in self.updater_config.iteritems( ) }
            self.updater_configs[ p ] = d
```

- 单步更新权重

在训练器进行单步更新时，会根据批量大小进行数据采样，然后调用学习模型的 loss 函数获取当前损失值以及梯度，再将梯度值与权重传递给更新器 updater，更新器返回更新后的权重 next_w，最后 trainer 将其替换为自身权重即可，请参考下列代码。单步更新属于训练器的私有行为，因此外界无法调用。

trainer 单步更新代码块：

```
def _step( self ) :
        """
        执行单步梯度更新。
        """
        # 采样批量数据。
        num_train = self.X_train.shape[ 0 ]
        batch_mask = np.random.choice( num_train, self.batch_size )
        X_batch = self.X_train[ batch_mask ]
        y_batch = self.y_train[ batch_mask ]
        # 计算损失值及梯度。
```

```
        loss, grads = self.model.loss( X_batch, y_batch )
        self.loss_history.append( loss )
        # 更新参数。
        for p, w in self.model.params.iteritems( ) :
          dw = grads[ p ]
          config = self.updater_configs[ p ]
          next_w, next_config = self.update_rule( w, dw, config )
          self.model.params[ p ] = next_w
          self.updater_configs[ p ] = next_config
```

- 模型精度检验

模型的验证部分相信你已经非常熟悉了，我们将该部分内容封装在了 trainer.check_accuracy( )函数中，直接使用即可。

检验训练精度代码模块：

```
def check_accuracy( self, X, y, num_samples = None, batch_size = 100 ) :
    """
   根据提供的数据检验精度。若数据集过大，可进行采样测试。
    Inputs:
    - X:形状为( N, d_1, ..., d_k )的数据。
    - y:形状为 （N, )的数据类标。
    - num_samples:采样次数。
    - batch_size:批量数据大小。
    Returns:
    - acc: 测试数据正确率。
    """
    # 对数据进行采样。
    N = X.shape[ 0 ]
    if num_samples is not None and N > num_samples :
      mask = np.random.choice( N, num_samples )
      N = num_samples
      X = X[ mask ]
      y = y[ mask ]
    # 计算精度。
    num_batches = N / batch_size
    if N % batch_size != 0:
      num_batches += 1
    y_pred = [ ]
    for i in xrange( num_batches ) :
      start = i * batch_size
```

```
        end = ( i + 1) * batch_size
        scores = self.model.loss( X[ start : end ] )
        y_pred.append( np.argmax( scores, axis = 1 ) )
    y_pred = np.hstack( y_pred )
    acc = np.mean( y_pred == y )
    return acc
```

- 模型训练

train()函数模块你也非常熟悉了，这里我们只是把全连接网络的 train()函数移植到了训练器中。阅读下列函数代码块，确保你熟悉并了解整个训练流程。

train 函数代码模块：

```
def train( self ) :
    """
    根据配置训练模型。
    """
    num_train = self.X_train.shape[ 0 ]
    iterations_per_epoch = max( num_train / self.batch_size, 1 )
    num_iterations = self.num_epochs * iterations_per_epoch
    for t in xrange( num_iterations ) :
        self._step( )
        # 打印损失值。
        if self.verbose and t % self.print_every == 0:
            print '(迭代  %d / %d)  损失值: %f' % (
                        t + 1, num_iterations, self.loss_history[ -1 ] )
        # 更新学习率。
        epoch_end = ( t + 1) % iterations_per_epoch == 0
        if epoch_end :
            self.epoch += 1
            for k in self.updater_configs :
                self.updater_configs[ k ][ 'learning_rate' ] * = self.lr_decay
        # 在训练的开始、末尾，每一轮训练周期检验模型精度。
        first_it = ( t == 0 )
        last_it = ( t == num_iterations + 1 )
        if first_it or last_it or epoch_end :
            train_acc = self.check_accuracy( self.X_train, self.y_train,
                                                num_samples = 1000 )
            val_acc = self.check_accuracy( self.X_val, self.y_val )
            self.train_acc_history.append( train_acc )
            self.val_acc_history.append( val_acc )
```

```
            if self.verbose :
              print '(周期  %d / %d)  训练精度: %f;  验证精度: %f' % (
                        self.epoch, self.num_epochs, train_acc, val_acc )
            # 记录最佳模型。
            if val_acc > self.best_val_acc :
              self.best_val_acc = val_acc
              self.best_params = { }
              for k, v in self.model.params.iteritems( ) :
                self.best_params[ k ] = v.copy( )
        # 训练结束后返回最佳模型。
        self.model.params = self.best_params
```

4.7.5　解耦更新器 updater

updater 负责更新神经网络的权重，其传入参数有神经网络的权重 *w*、当前权重的梯度 *dw* 及相应的更新配置。比如在随机梯度下降 sgd 中，我们使用学习率和当前权重梯度更新权重。该模块将在第 5 章中重点讲解，现在我们仅提供了最原始的 sgd 算法。

Updater 代码模块：

```
#-*- coding: utf-8 -*-
import numpy as np
"""
频繁使用的神经网络一阶梯度更新规则。每次更新接收：当前的网络权重，
训练获得的梯度及相关配置进行权重更新。
def update( w, dw, config = None ) :
Inputs:
  - w:当前权重。
  - dw:  与权重形状相同的梯度。
  - config:  字典型超参数配置，比如学习率、动量值等。
如果更新规则需要用到缓存，在配置中需要保存相应的缓存。
Returns:
  - next_w:  更新后的权重。
  - config:  更新规则相应的配置。
"""
def sgd( w, dw, config = None ) :
  """
  随机梯度下降更新规则。
  config :
  - learning_rate:  学习率。
  """
```

```
if config is None: config = { }
config.setdefault( 'learning_rate', 1e-2 )
w -= config[ 'learning_rate' ] * dw
return w, config
```

- 训练神经网络

解耦神经网络后，接下来我们测试无 Dropout 情况下的全连接网络，运行下列代码模块，无 Dropout 神经网络训练可视化效果如图 4-5 所示。

训练神经网络代码模块：

```
model = None
trainer = None
D, H, C, std, r = 3*32*32, 200, 10, 1e-2, 0.6
model = FullyConnectedNet( input_dim = D,   hidden_dims = [ H ], num_classes = C,   weight_scale = std )
trainer = Trainer( model, data, update_rule = 'sgd',
                    updater_config = { 'learning_rate' : 1e-3, },
                    lr_decay = 0.95, num_epochs = 20,
batch_size = 200, print_every = 200 )
trainer.train( )
```

神经网络解耦训练结果：

```
(迭代  1 / 4900)  损失值: 4.680202
(周期  0 / 20)  训练精度: 0.155000;  验证精度: 0.163000
(迭代  201 / 4900)  损失值: 1.954245
(周期  1 / 20)  训练精度: 0.335000;  验证精度: 0.361000
(迭代  401 / 4900)  损失值: 1.626711
(周期  2 / 20)  训练精度: 0.448000;  验证精度: 0.416000
(迭代  601 / 4900)  损失值: 1.764400
                        ……
(周期  18 / 20)  训练精度: 0.647000;  验证精度: 0.517000
(迭代  4601 / 4900)  损失值: 1.024862
(周期  19 / 20)  训练精度: 0.641000;  验证精度: 0.498000
(迭代  4801 / 4900)  损失值: 0.904784
(周期  20 / 20)  训练精度: 0.644000;  验证精度: 0.515000
```

可视化训练结果：

```
# 可视化训练/验证结果。
plt.subplot( 2, 1, 1 )
plt.title( 'Training loss' )
```

```
plt.plot( trainer.loss_history, 'o' )
plt.xlabel( 'Iteration' )
plt.subplot( 2, 1, 2 )
plt.title( 'Accuracy' )
plt.plot( trainer.train_acc_history, '-o', label = 'train' )
plt.plot( trainer.val_acc_history, '-o', label = 'val' )
plt.plot( [ 0.5 ] * len( trainer.val_acc_history ), 'k--' )
plt.xlabel( 'Epoch' )
plt.legend( loc = 'lower right' )
plt.gcf( ).set_size_inches( 15, 12 )
plt.show( )
```

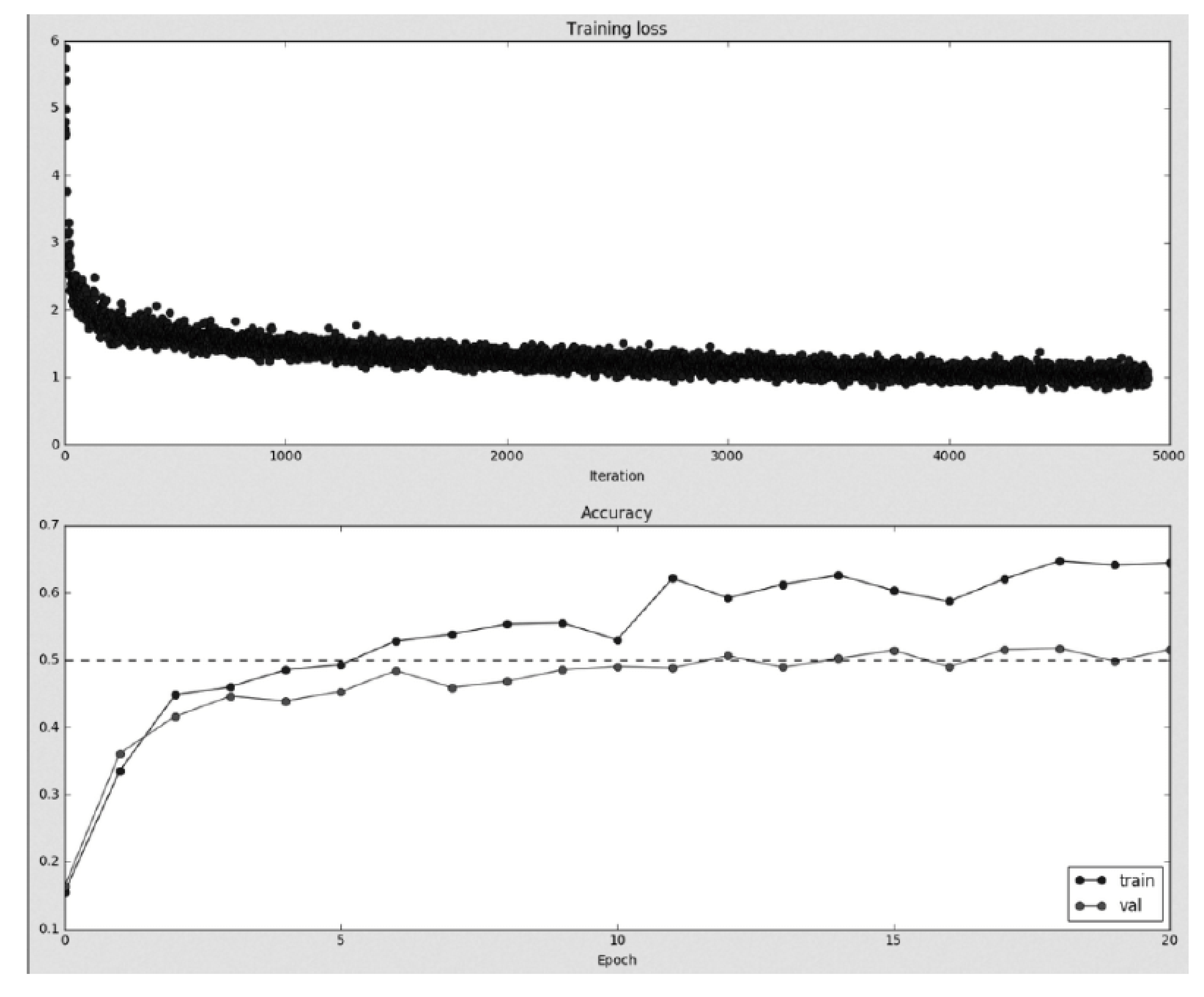


图 4-5　无 Dropout 神经网络训练结果

4.7.6　正则化实验

Dropout 是一种高效且简单的正则化措施，接下来我们使用 0、0.3 和 0.7 神经元激活概率，测试在少量数据时的网络性能。运行下列代码块，可视化效果如图 4-6 所示。

正则化实验代码模块：

```
num_train = 500
small_data = {
  'X_train': data[ 'X_train' ][ : num_train ],
  'y_train': data[ 'y_train' ][ : num_train ],
```

```
    'X_val': data[ 'X_val' ],
    'y_val': data[ 'y_val' ],
}
solvers = { }
dropout_choices = [ 0, 0.3,0.7 ]
for dropout in dropout_choices :
    model = FullyConnectedNet( hidden_dims = [ 600 ], dropout = dropout )
    print "dropout 激活概率(0 表示不使用 dropout)%f:" % dropout
    trainer = Trainer( model, small_data,
                        num_epochs = 30, batch_size = 100,
                        update_rule = 'sgd',
                        updater_config = { 'learning_rate': 5e-4, },
                        verbose = True, print_every = 200 )
    trainer.train( )
    solvers[ dropout ] = trainer
```

```
可能的运行结果：
dropout 激活概率(0 表示不使用 dropout)0.000000:
(迭代 1 / 150) 损失值: 8.925743
(周期 0 / 30) 训练精度: 0.164000; 验证精度: 0.160000
(周期 1 / 30) 训练精度: 0.262000; 验证精度: 0.199000
                ……
(周期 29 / 30) 训练精度: 1.000000; 验证精度: 0.298000
(周期 30 / 30) 训练精度: 1.000000; 验证精度: 0.299000
dropout 激活概率(0 表示不使用 dropout)0.300000:
(迭代 1 / 150) 损失值: 17.888175
(周期 0 / 30) 训练精度: 0.166000; 验证精度: 0.143000
(周期 1 / 30) 训练精度: 0.290000; 验证精度: 0.189000
                ……
(周期 29 / 30) 训练精度: 0.978000; 验证精度: 0.323000
(周期 30 / 30) 训练精度: 0.986000; 验证精度: 0.315000
dropout 激活概率(0 表示不使用 dropout)0.700000:
(迭代 1 / 150) 损失值: 10.675305
(周期 0 / 30) 训练精度: 0.160000; 验证精度: 0.140000
(周期 1 / 30) 训练精度: 0.298000; 验证精度: 0.204000
(周期 2 / 30) 训练精度: 0.398000; 验证精度: 0.228000
                ……
(周期 29 / 30) 训练精度: 1.000000; 验证精度: 0.289000
(周期 30 / 30) 训练精度: 1.000000; 验证精度: 0.283000
```

可视化训练结果代码块：

```
train_accs = [ ]
val_accs = [ ]
for dropout in dropout_choices :
    solver = solvers[ dropout ]
    train_accs.append( solver.train_acc_history[ -1 ] )
    val_accs.append( solver.val_acc_history[ -1 ] )
plt.subplot( 3, 1, 1 )
for dropout in dropout_choices :
    plt.plot( solvers[ dropout ].train_acc_history, 'o', label = '%.2f dropout' % dropout )
plt.title( 'Train accuracy' )
plt.xlabel( 'Epoch' )
plt.ylabel( 'Accuracy' )
plt.legend( ncol = 2, loc = 'lower right' )
plt.subplot( 3, 1, 2 )
for dropout in dropout_choices :
    plt.plot( solvers[ dropout ].val_acc_history, 'o', label = '%.2f dropout' % dropout )
plt.title( 'Val accuracy' )
plt.xlabel( 'Epoch' )
plt.ylabel( 'Accuracy' )
plt.legend( ncol = 2, loc = 'lower right' )
plt.gcf( ).set_size_inches( 15, 15 )
plt.show( )
```

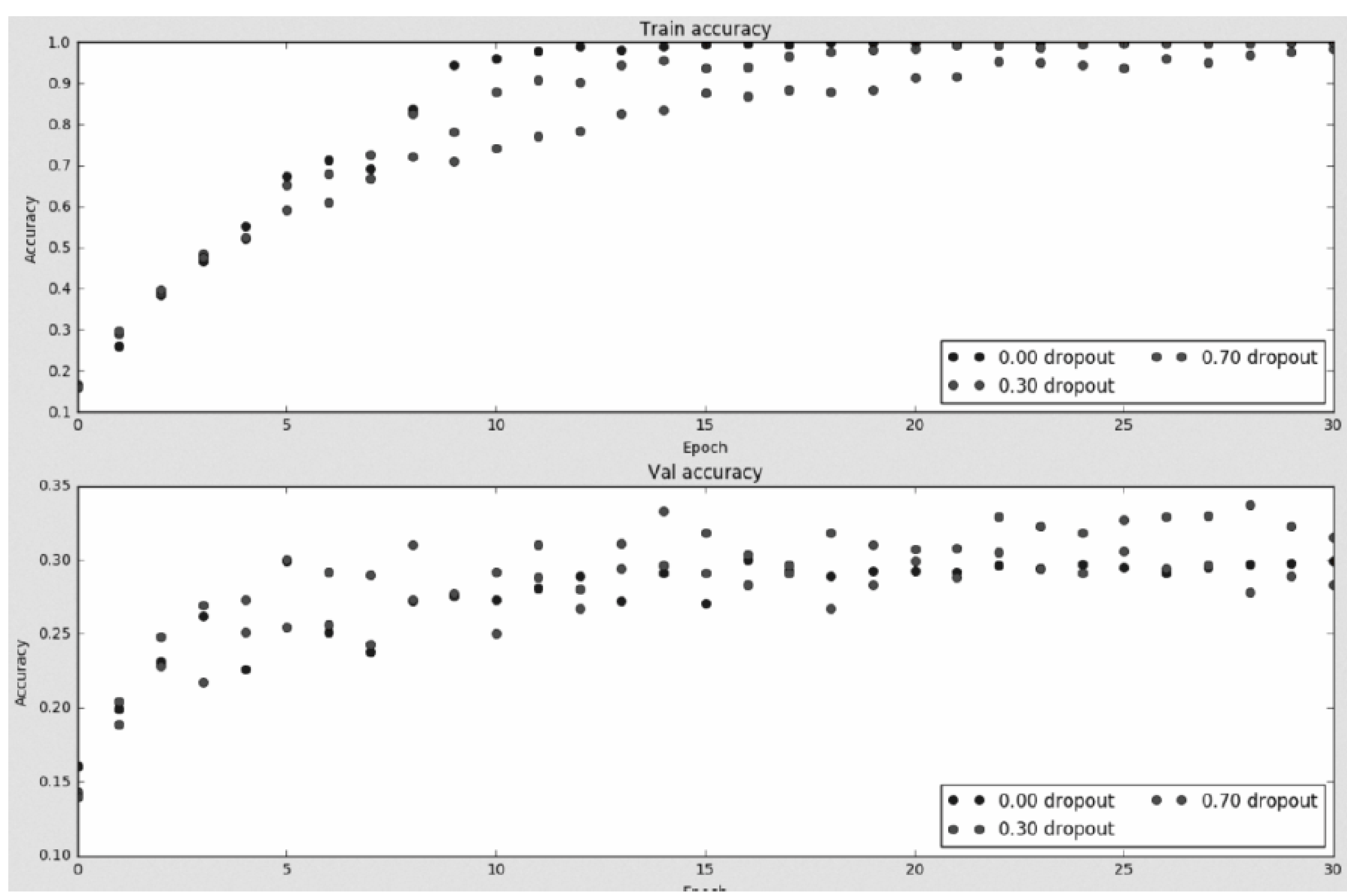


图 4-6　Dropout 正则化实验

从图 4-6 中，我们可以清晰地看出，当不使用 Dropout 时，网络在训练阶段很快就产生了过拟合现象，并且较低的神经元激活概率(0.3)可以有效地缓解过拟合现象，并且在验证时也拥有最佳验证精度。

4.8 参考代码

dropout_forward 函数代码块：

```
def dropout_forward( x, dropout_param ) :
    p, mode = dropout_param[ 'p' ], dropout_param[ 'mode' ]
    if 'seed' in dropout_param :
        np.random.seed( dropout_param[ 'seed' ] )
    mask = None
    out = None
    if mode == 'train' :
        mask = ( np.random.rand( * x.shape ) < p ) / p
        out = x * mask
    elif mode == 'test' :
        out = x
    cache = ( dropout_param, mask )
    out = out.astype( x.dtype, copy = False )
    return out, cache
```

dropout_backward 函数代码块：

```
def dropout_backward( dout, cache ) :
    dropout_param, mask = cache
    mode = dropout_param[ 'mode' ]
    dx = None
    if mode == 'train':
        dx = dout * mask
    elif mode == 'test':
        dx = dout
    return dx
```

affine_relu_dropout_forward 函数代码块：

```
def affine_relu_dropout_forward( x, w, b, dropout_param ) :
    out_dropout = None
    cache = None
    out_affine,   cache_affine = affine_forward( x, w, b )
    out_relu, cache_relu = relu_forward( out_affine )
```

```
    out_dropout, cache_dropout = dropout_forward( out_relu, dropout_param )
    cache = ( cache_affine, cache_relu, cache_dropout )
    return out_dropout, cache
```

affine_relu_dropout_backward 函数代码块：

```
def affine_relu_dropout_backward ( dout, cache ) :
    cache_affine, cache_relu, cache_dropout = cache
    dx, dw, db = None, None, None
    ddropout = dropout_backward( dout, cache_dropout )
    drelu = relu_backward( ddropout, cache_relu )
    dx, dw, db = affine_backward( drelu, cache_affine )
    return dx, dw, db
```

Loss 函数代码块：

```
def loss ( self, X, y = None ) :
    mode = 'test' if y is None else 'train'
    if self.dropout_param is not None:
        self.dropout_param[ 'mode' ] = mode
    scores = None
    outs, cache = { }, { }
    outs[ 0 ] = X
    num_h = self.num_layers - 1
    for i in xrange( num_h ) :
        if self.use_dropout :
            outs[ i + 1 ], cache[ i +1] = affine_relu_dropout_forward(
                outs[ i ], self.params[ 'W' + str( i + 1 ) ],
                self.params[ 'b' + str( i + 1 ) ], self.dropout_param )
        else :
            outs[ i + 1], cache[ i + 1 ] = affine_relu_forward(
                outs[ i ], self.params[ 'W' + str( i + 1 ) ],
                self.params[ 'b' + str( i + 1 ) ] )
    scores,cache[ num_h + 1 ] = affine_forward(
        outs[ num_h ], self.params[ 'W' + str( num_h + 1 ) ],
        self.params[ 'b' + str( num_h + 1 ) ] )
    if mode == 'test' :
        return scores
    loss, grads = 0.0, { }
    dout = { }
    loss, dy = softmax_loss( scores, y )
```

```
h = self.num_layers -1
for i in xrange( self.num_layers ) :
    loss + = 0.5 * self.reg * ( np.sum( self.params[ 'W' + str ( i + 1 ) ] *
                                    self.params[ 'W' + str( i + 1 ) ] ) )
dout[ h ], grads[ 'W' + str( h + 1 ) ], grads[ 'b' + str( h + 1 ) ] = affine_backward( dy, cache[ h + 1 ] )
grads[ 'W' + str( h + 1 ) ] + = self.reg * self.params [ 'W' + str( h + 1 ) ]
for i in xrange( h ) :
    if self.use_dropout :
        dout[ h - 1 - i ], grads[ 'W' + str( h – i ) ], grads[ 'b' + str( h - i ) ] =
                                affine_relu_dropout_backward( dout[ h – i ], cache[ h – i ] )
    else :
        dout[ h - i - 1 ], grads[ 'W' + str( h – i ) ], grads[ 'b' + str( h – i ) ]    =
                                        affine_relu_backward( dout[ h – i ], cache[ h – i ] )
    grads[ 'W' + str( h – i ) ] + = self.reg * self.params[ 'W' + str( h – i ) ]
return loss, grads
```

4.9　参考文献

[1] Tibshirani, R. J. (1996). Regression shrinkage and selection via the LASSO. J R Stat Soc B. Journal of the Royal Statistical Society, 58, 267-288.

[2] Saunders, C., Gammerman, A., & Vovk, V. (1999). Ridge Regression Learning Algorithm in Dual Variables. Paper presented at the Proc. of the 15th Int. Conf. on Machine Learning ICML-98, Madison-Wisconsin.

[3] Ng, A. Y. (2004). Feature selection, L 1 vs. L 2 regularization, and rotational invariance. Icml, 19(5), 379-387.

[4] Mozer, M. C. (1993). Proceedings of the 1993 Connectionist Models Summer School: L. Erlbaum Associates.

[5] Lecun, Y., & Bengio, Y. (1998). Convolutional networks for images, speech, and time series: MIT Press.

[6] Cui, X., Goel, V., & Kingsbury, B. (2014). Data augmentation for deep convolutional neural network acoustic modeling. Paper presented at the ICASSP 2014 - 2014 IEEE International Conference on Acoustics, Speech and Signal Processing.

[7] Sietsma, J., & Dow, R. J. F. (1991). Creating artificial neural networks that generalize. Neural Networks, 4(1), 67-79.

[8] Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., & Manzagol, P. A. (2010). Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion. Journal of Machine Learning Research, 11(12), 3371-3408.

[9] Bishop, C. M. (1995). Training with Noise is Equivalent to Tikhonov Regularization. Neural Computation, 7(1), 108-116.

[10] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15(1), 1929-1958.

[11] Schmidhuber, J., & Informatik, F. F. (1999). Simplifying Neural Nets By Discovering Flat Minima.

[12] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2015). Rethinking the Inception Architecture for Computer Vision. Computer Science, 2818-2826.

[13] Schölkopf, B., Platt, J., & Hofmann, T. (2006). Efficient Learning of Sparse Representations with an Energy-Based Model. Paper presented at the Advances in neural information processing systems.

[14] Goodfellow, I. J., Le, Q. V., Saxe, A. M., Lee, H., & Ng, A. Y. (2009). Measuring invariances in deep networks. Paper presented at the Advances in Neural Information Processing Systems 22: Conference on Neural Information Processing Systems 2009. Proceedings of A Meeting Held 7-10 December 2009, Vancouver, British Columbia, Canada.

[15] Larochelle, H., & Bengio, Y. (2008). Classification using discriminative restricted Boltzmann machines. Paper presented at the International Conference.

[16] Caruana, R., Lawrence, S., & Giles, L. (2001). Overfitting in neural nets: backpropagation, conjugate gradient, and early stopping. Paper presented at the International Conference on Neural Information Processing Systems.

[17] SJÖBERG, J., & LJUNG, L. (1995). Overtraining, regularization and searching for a minimum, with application to neural networks. International Journal of Control, 62(6), 1391-1407.

[18] Breiman, L. (1996). Bagging Predictors. Machine Learning, 24(2), 123-140.

[19] Dietterich, T. G. (2000). Ensemble Methods in Machine Learning. 1857(1), 1-15.

[20] Freund, Y. (1996). Experiments with a new boosting algorithm. Proceedings of Icml, 13, 148-156.

第 5 章

深度学习优化

不知不觉中我们已经走过了前 4 章的内容，开始时你可能满怀期待地阅读本书。当编写完第一个深层神经网络时，可能也激动万分，感觉自己马上就能改变世界了。但随后的训练，或许让你感到绝望。训练深层神经网络是非常困难的，暗含大量的训练技巧，需要令人眼花缭乱的超参数配置种类，还可能经常会面临奇怪的数值溢出问题。为什么优化深度学习那么困难呢？

这就如同孤独的你，在迷雾中寻找下山的路。你可能需要面对大量的局部最优、鞍点、悬崖等客观存在的困难，也可能会面对不精确梯度（数据不准确），梯度消失等自身缺陷问题。稍有不慎，你就可能迷失自己，跌入万丈深渊。为了解救绝望的你，本章我们将介绍一些深度学习常用的优化手段，希望能够帮助你训练神经网络，让你再次重燃热情。

在本章中，我们首先会解释神经网络优化困难的一些具体原因，然后会介绍一些基本的优化算法，如 SGD、Momentum、AdaGrad、RMSProp 及 Adam 学习方法。在此之后，我们还会介绍神经网络的**参数初始化**策略，而在最后，我们将重点介绍深度学习的小明星——**批量归一化**（Batch Normalization）。

在本章的编程练习中，我们会逐个实现 Momentum、RMSProp 及 Adam 算法，并简单地比较它们的性能。最后，我们会将重点放在批量归一化的编程练习上。虽然其理解很容易，但实际编程可能让你有点痛苦，希望我们能够一起圆满地解决这些困难。

5.1　神经网络优化困难

最优化是一个极度困难的问题，在机器学习中，通常需要小心翼翼地设计目标函数及其约束，然后还要确保面对的优化问题是一个凸函数，之后才能进行机器学习。但非常遗憾，在神经网络中我们必须经常面对非凸函数的优化问题。本小节我们将介绍一些深度学习所面临的优化挑战，这其中包括了**局部最优**、**鞍点**、**梯度悬崖**及**梯度消失**问题。

5.1.1　局部最优

凸函数最显著的特征是能够找到一个局部最优解，并且此解是全局的最优解。但有时我们也并不执着于寻找最优的某一点，在某些凸函数的底部可能是一个犹如盆地的平坦区域，在该平坦区域取得的任何值也是可以接受的。

但如图 5-1 所示，在非凸函数中，有可能含有多个局部最优解。特别是在深度神经网络中，局部最优解的数量就更多了。就好像我们寻找下山的路，只能走一步看一步。如果局部最优很多，那我们很可能就迷失在了半山腰。

但局部最优也并非想象得那样可怕，对于机器学习而言，我们其实不太害怕局部最优解，我们害怕的是找到的解和全局最优相差很大，并且机器学习也并非只是一个纯粹的优化问题，在已知数据中找到最优解也不是机器学习的目的。在实践中我们发现，最优解附近的解其泛化性能通常也要比最优解要好。

多年前，多名数学研究者都相信局部最优会是神经网络中非常普遍的灾难性问题[1]。而如今，这也不是一个显著问题，虽然对于该领域的研究仍然火热，但专家们现在开始怀疑，对于大规模的神经网络而言，大多数局部最优都有一个比较低的损失值[2]，并且寻找真实的全局最优也不是一个很重要的问题，重要的是在参数空间中找到一个相对较低的局部最优值。

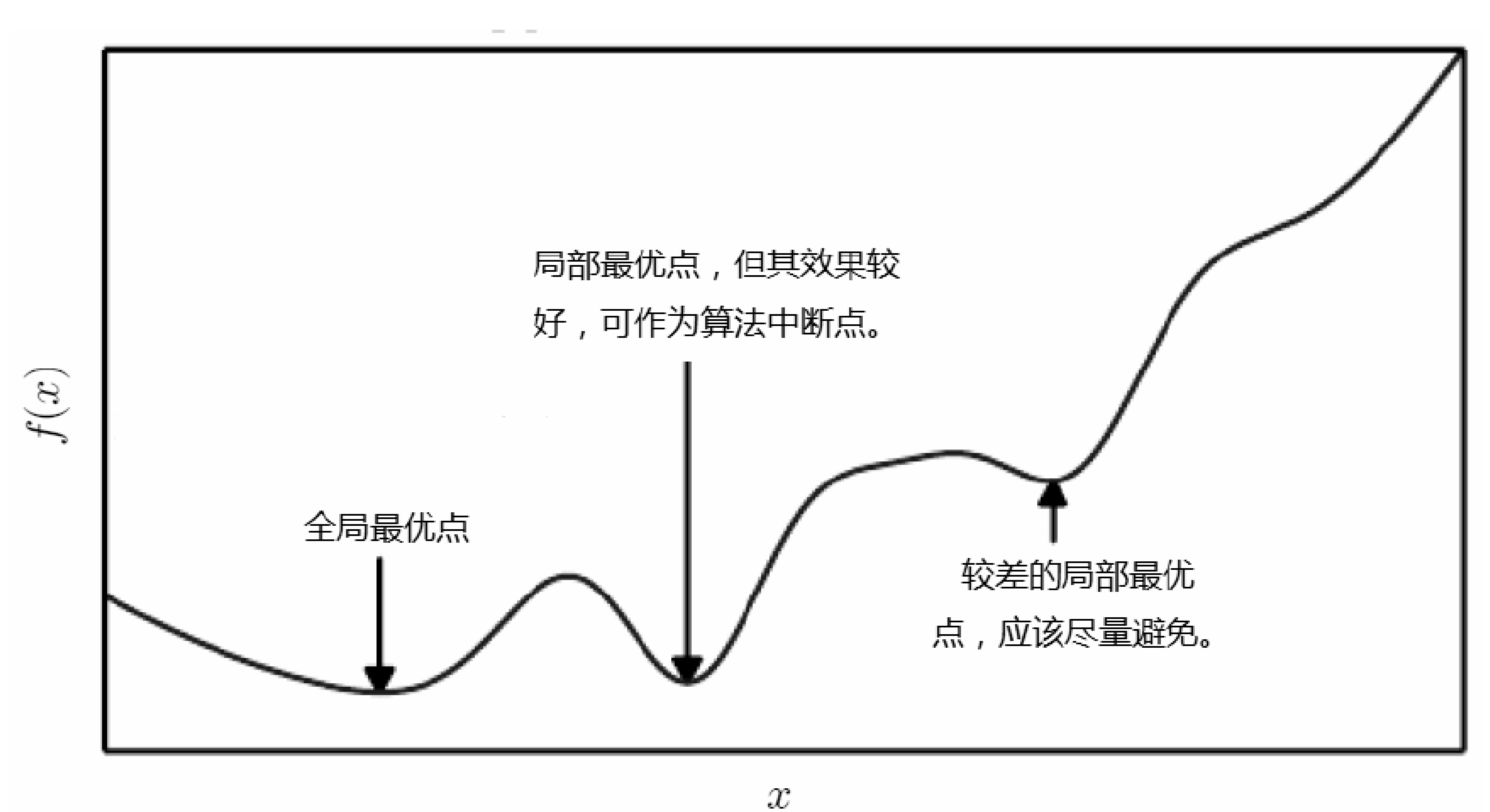


图 5-1　函数局部最优解示意图

5.1.2 鞍点

对于高维数据来说，局部最优可能已不是非常严重的问题，因为存在着一个更突出的问题——**鞍点**（saddle point）。如图 5-2 所示，鞍点就像是两座山峰的中间区域，该区域不是局部最优值，但该区域十分平坦，换言之就是**梯度几乎为零**。就好像你在大雾中找到一块平地，你环顾四周都无法确定一条路。对于鞍点和局部最优，可以想象成是丢硬币的过程：硬币全为正面就为局部最优解，有部分为正面就为鞍点，硬币的个数就是数据的维度，如果维度越高，那么出现局部最优的概率也就越低，但鞍点的数目，却是成指数增长[3]。

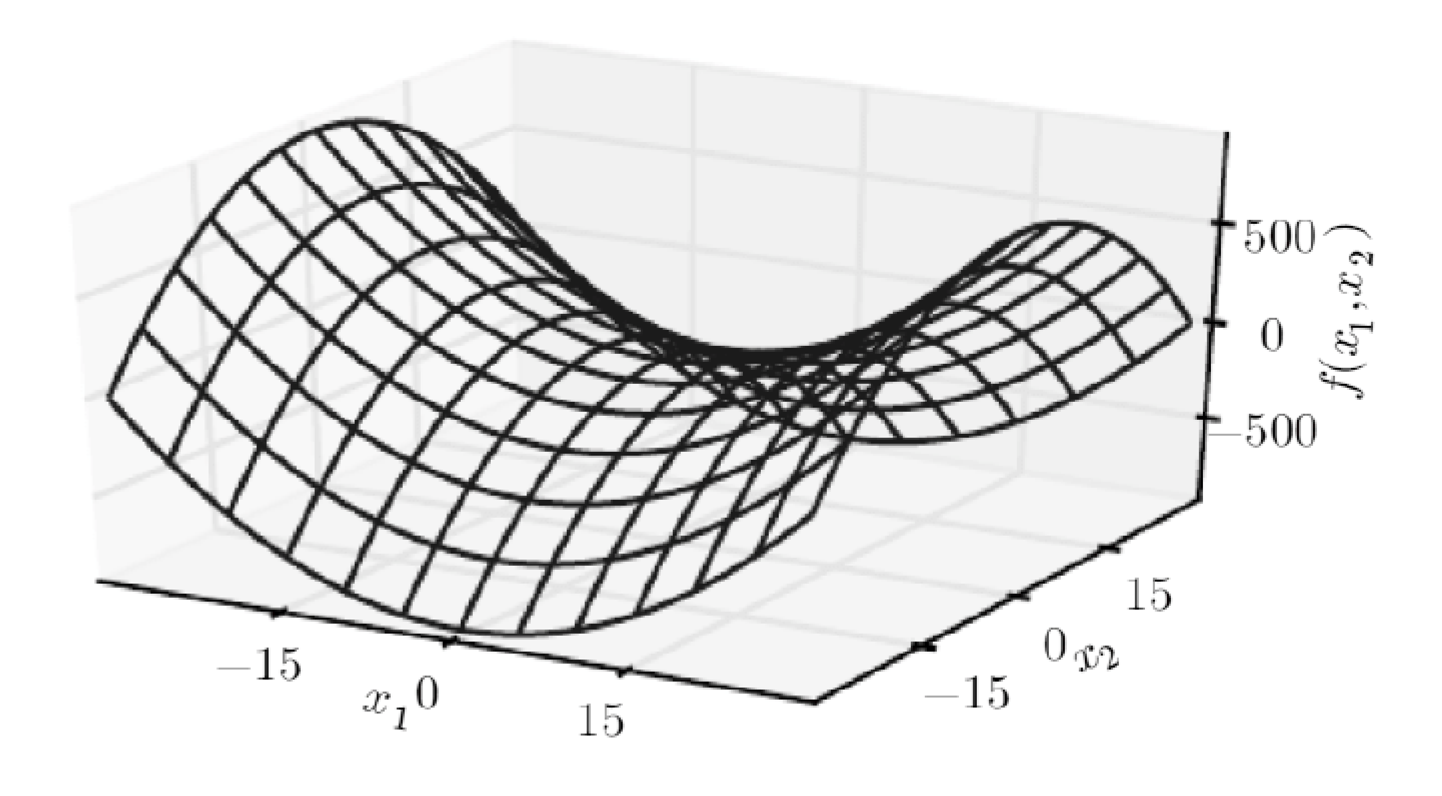


图 5-2 鞍点示意图

5.1.3 梯度悬崖

多层神经网络还经常会有极度陡峭的区域，就如同悬崖一般，如图 5-3 所示，高度非线性的深度神经网络或者循环神经网络，在参数空间中常常含有尖锐的非线性，这导致了某些区域可能会产生非常高的梯度[4]。当参数靠近这一悬崖区域，高梯度会将参数弹射到很远的地方，很可能导致原本的优化工作半途而废。

需要注意的是，由于循环神经网络涉及在多个时间段内相乘，因此梯度悬崖在递归神经网络中十分频繁，特别是处理较长的时序序列时，该问题会变得异常令人头疼。

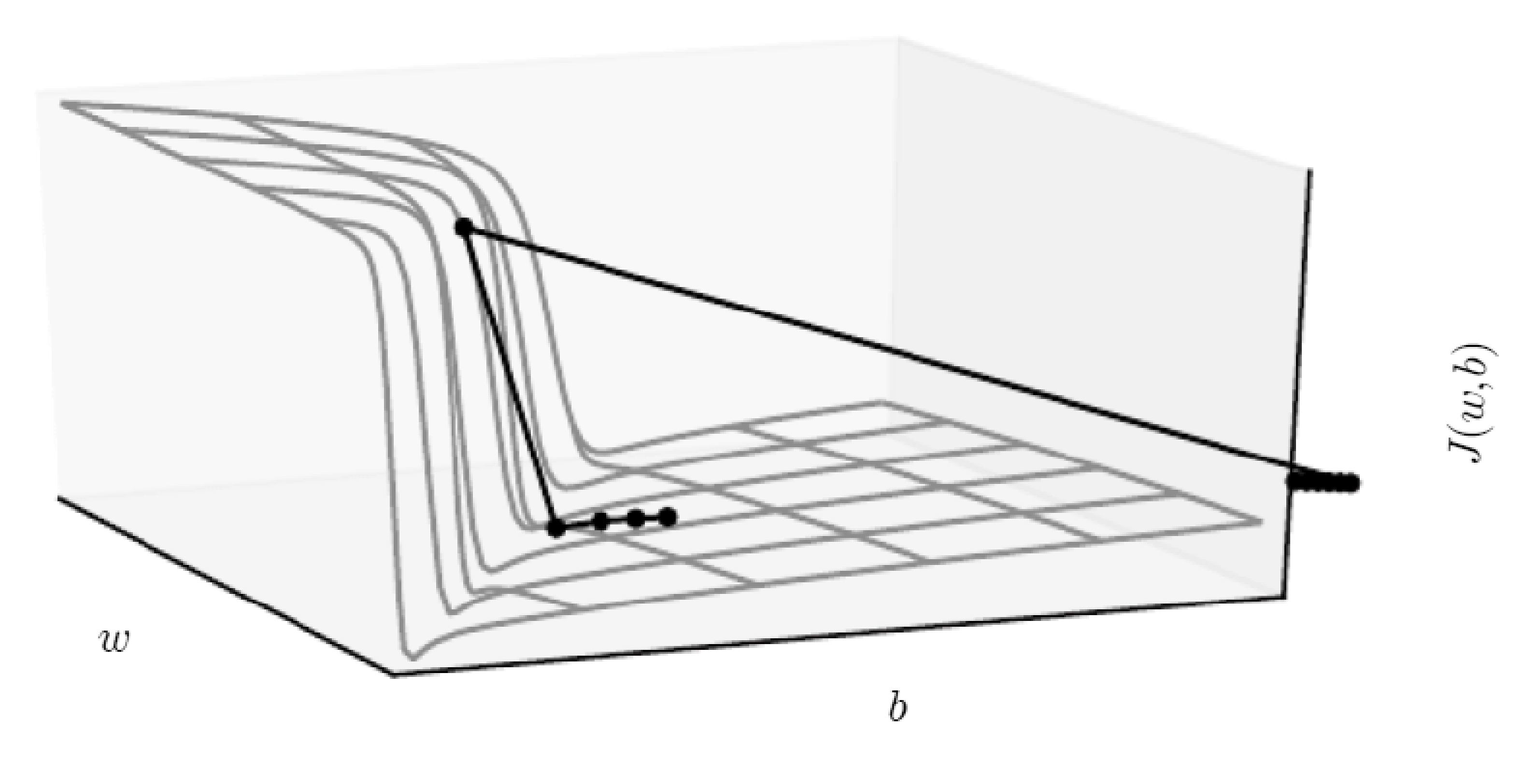


图 5-3 梯度悬崖示意图

5.1.4　梯度消失或梯度爆炸

深度学习最大的特点是其网络非常深（层数非常多），深度提升了模型的复杂度与能力，但也导致了深度学习中的一大难题，那就是**梯度消失问题**（Vanishing Gradient Problem）[5]。这个问题也是曾经导致深度学习只能被称为“（浅层）神经网络”的最主要原因。该问题随着 ReLU 单元的使用而得到了极大的缓解[6]，但在一些特定的神经网络中，如**循环神经网络**（RNN），该问题依然严重。

梯度为什么会消失呢？这其实是使用链式求导法则所引起的。如图 5-4 所示，为一个极度简化的 4 层单神经元的神经网络。根据 BP 算法进行计算，第一层的权重如式（5.1）所示。

$$\frac{\partial L}{\partial w_1}=(y-f_4(a_4))f_4'(a_4)w_3f_3'(a_3)w_2f_2'(a_2)x \tag{5.1}$$

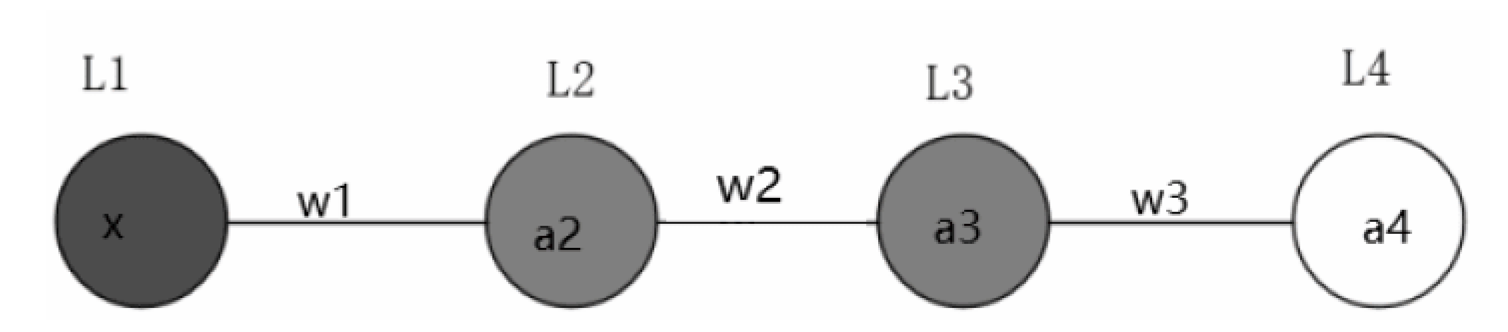


图 5-4　4 层单神经元神经网络示意图

最关键的问题就出现在激活函数的导数中，在我们的极简版 4 层神经网络中，第一层权重的梯度大约要乘以后三层激活函数的导数，如果我们使用 Sigmoid 神经元，该神经元的导函数的图像如图 5-5 所示，取值范围为(0,0.25]。即使我们都取其最大值，那经过三层反向传导之后也仅仅只有 0.015625。因此在深层网络中，即使网络预测产生了很大的误差，但底层的神经元依然没有得到足够的误差修正。

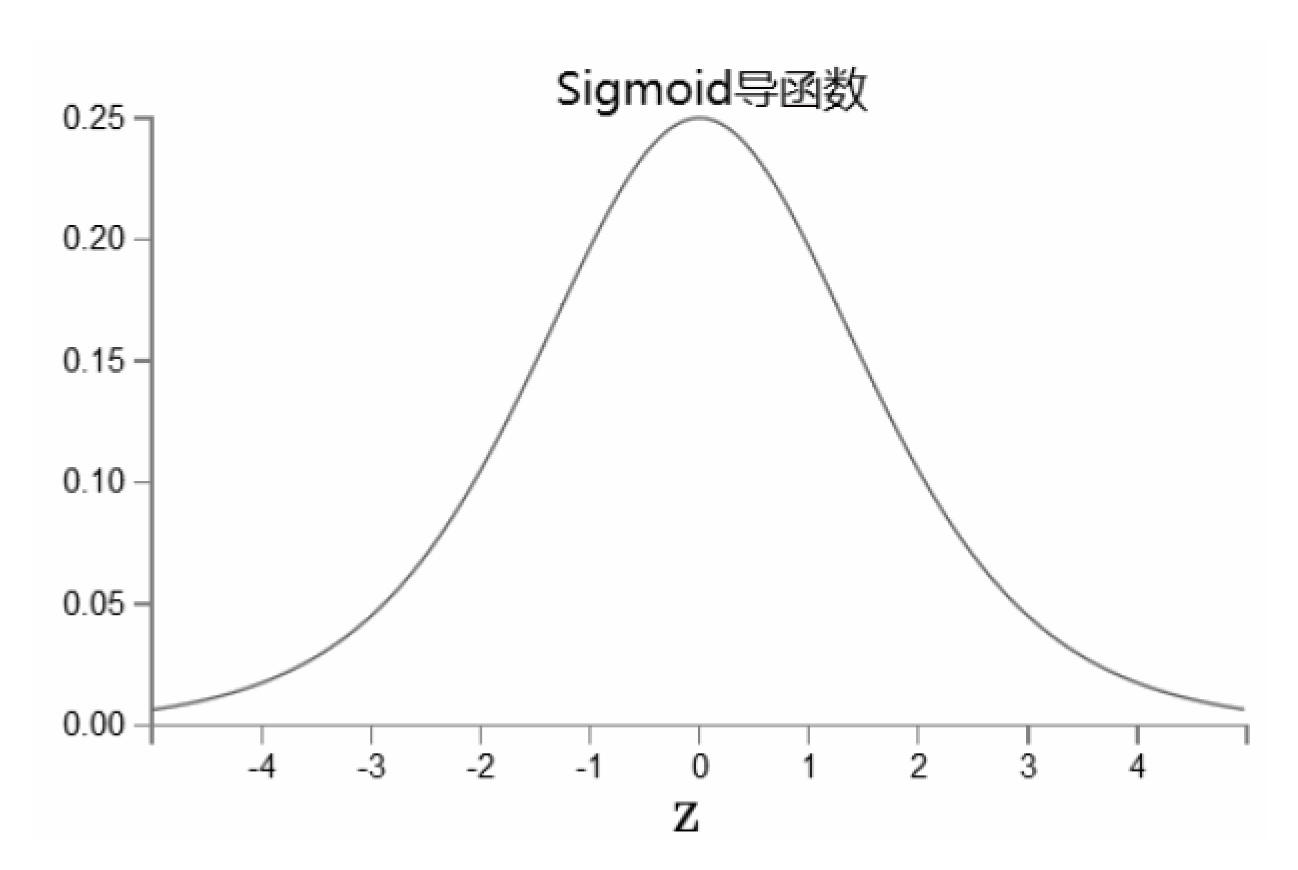


图 5-5　Sigmoid 神经元导函数图像

那什么又是**梯度爆炸问题**（Exploding Gradient Problem）[7]呢？如式（5.1）所示，本层权重的梯度可以简化为上层各神经元的梯度与其权重的乘积，如果上层的权重过大，当经过传递后，本层的梯度就会变得异常巨大，造成梯度非常不稳定。但相比于梯度消失，梯度爆炸问题比较容易解决，并且发生的情况也不频繁。

5.1.5 梯度不精确

大多数优化算法最原始的动机都是试图获取代价函数对应的精确梯度，从而优化学习器。但在实践中，我们经常使用含有噪声的梯度进行优化。比如梯度下降法需要遍历所有训练数据后计算出平均梯度，然后才修改网络，但这种方法在数据较大时训练速度太慢，因此深度学习通常会进行数据采样，使用最小批量梯度下降学习方法进行网络训练，甚至在极端情况下还会使用随机梯度下降（一次采样一条数据）进行网络训练。

这种不精确的梯度，也就导致了训练的稳定性较差，但梯度不精确有时也可以看作是防止过拟合以及逃离局部最优或鞍点的方法。机器学习终究不是一个最优化问题，但其却要依靠优化手段来完成机器学习任务。具体问题，还需要根据实际需求进行思考。

5.1.6 优化理论的局限性

一些理论结果显示，很多针对神经网络而设计的优化算法有着局限性[8]，但在实践中，这些理论结果却很少对神经网络产生影响。这也是相比于其他机器学习算法而言，神经网络更像是一个黑盒。神经网络中存在着大量的训练技巧，这也使得训练神经网络更像是艺术而非科学。

在神经网络的训练中，我们通常不关心能否找到精确的全局最优解，我们仅仅是去降低代价函数的值，使其能够获得较好的泛化性能。但我们并不是不想获得全局最优解，只是理论分析神经网络算法是一个极其困难的任务。总而言之，深度学习是实践的产物，还缺乏强有力的理论支持，很多科研人员仍然对其保持着怀疑态度，如何理智地评估深度学习算法性能边界仍然是机器学习中一个重要的目标。

5.2 随机梯度下降

随机梯度下降（Stochastic Gradient Descent，SGD）[9]及其变异，可以算是深度学习中使用最广泛的优化算法了。早在本书的第 2 章中我们就简单地介绍过该算法，在本小节中我们将更加全面地介绍该算法，如果你已经非常熟悉此部分的内容，可以直接跳到 5.3 节动量学习法中进行学习。

如算法 5.1 所示，是一个标准的随机梯度下降训练过程。其实就是选择一条数据，就训练一条数据，然后修改一次权重。SGD 算法在训练过程中很有可能选择被错误标记的数据，或者与正常数据差异很大的数据进行训练，那么使用此数据求得的梯度就会有很大的偏差，因此 SGD 在训练过程中会出现很强的随机现象。

算法 5.1：随机梯度下降算法
给定数据集 $X=\{x^{(1)},x^{(2)},\cdots,x^{(n)}\}$，数据集标记 $Y=\{y^{(1)},y^{(2)},\cdots,y^{(n)}\}$， 学习器 $f(x;w)$，学习率（步长）α。

For 迭代足够多次

{

1.随机选择数据：$\{x^{(j)}, y^{(j)}\}$

2.计算损失梯度：$\nabla w = \frac{\partial L(y^{(j)}, f(x^{(j)};w))}{\partial w}$

3.修改权重：$w_i = w_i - \alpha \nabla w$

}

为了防止这种随机性带来的危害，我们就多选几条数据，然后计算一下多条数据的平均错误，如式（5.2）所示。即使有某条数据存在严重缺陷，也会因为多条数据的中和而降低其错误的程度。就好像是一个人发现了错误，但并没有急着调整自己，而是再多看看其他的一些情况，然后综合地来调整自己。

$$\nabla w = \frac{1}{m}\sum_{j=1}^{m}\frac{\partial L(y^{(j)}, f(x^{(j)};w))}{\partial w} \tag{5.2}$$

在上述的算法中，学习率 α 是固定的值。但在实践中，我们往往需要通过训练次数的增长而逐渐地降低学习率。SGD 算法引入了源噪声，而这种噪声是个体数据的特殊性所造成的，因此，即使我们的算法到达了最优解附近，其噪声也不会消失，这也就形成在最优解附近振荡的现象。为了消除或缓解这种情况，我们就在靠近最优解周围尽可能地减少学习率。这就如同我们的成长一样，年轻时我们容易出错，我们也很乐于调整自己，反思自己。随着我们的成长，学习到的知识越来越多，但我们也会越来越固执。当人到中年，所见事物和自己想的不一样，我们很可能不是去学习新事物，而是去质疑那些“不一样”，否认新事物的发生，即使面对不得不调整自己的情况，也只是懒散地做出微调。

综上所述，我们将原本固定的学习率 α 设计为时间衰减的形式。初始时学习率较高，随着训练轮数的增加，学习率不断地减少。但我们也不希望学习率一直衰减，因此也会设置一个学习率的最低值。如式（5.3）所示，为训练 k 轮时第 i 轮学习率的取值。

$$\alpha_i = (1-\frac{i}{k})\alpha_0 + \frac{i}{k}b \tag{5.3}$$

学习率的取值需要反复的实验，通常最好的方式是通过代价函数的变化情况来监控学习率的变化曲线。但这与其说是科学，还不如说是一项艺术，许多针对学习率的调整方法都太过技巧性。当我们使用式（5.3）这样的线性策略来调整学习率时，通常需要进行上百轮的学习调整，因此 k 要大于一百，而 b 的取值可以粗略地设置为百分之一的初始学习率。现在的主要问题就是如何设置初始学习率，如果太大，学习曲线就会剧烈震荡；如果初始值太小，那学习过程就会非常缓慢，并且还会陷入一个比较高的代价函数区域。在实际测试中，学习率通常要作为超参数进行选择，并没有一个固定的取值方法。综上所述，如算法 5.2 所示，就为改进的随机梯度下降后的学习率衰减最小批量梯度下降训练法。

算法 5.2：学习率衰减最小批量梯度下降训练法
初始化： 给定数据集 $X=\{x^{(1)}, x^{(2)}, \cdots, x^{(n)}\}$，数据集标记 $Y=\{y^{(1)}, y^{(2)}, \cdots, y^{(n)}\}$， 随机采样 m 条数据，训练周期 k，学习率衰减最低值 b，学习器 $f(x;w)$，

初始学习率（步长）α_0。
训练：
For i <= k
{
1.随机采样 m 条数据：$\{(x^{(1)},y^{(1)})\cdots(x^{(m)},y^{(m)})\}$
2.计算采样数据平均损失值梯度：$\nabla w=\frac{1}{m}\sum_{j=1}^{m}\frac{\partial L(y^{(j)},f(x^{(j)};w))}{\partial w}$
3.计算衰减学习率：$a_i=(1-\frac{i}{k})a_0+\frac{i}{k}b$
4.修改网络权重：$w_i=w_i-\alpha\nabla w$
}

5.3 动量学习法

我们经常把梯度下降比拟成在迷雾中下山，而梯度下降法其实就是走一步，然后停下来看一步，然后再走一步。这种方法使用起来非常方便和有效，但当损失函数接近鞍点或局部最优点时，我们的梯度就会变得非常小，优化的过程也就会变得异常的缓慢。

我们可以将梯度理解成力，该力指引着我们向山谷前行，但我们并不是靠力前行，而靠的是速度，而力只是改变速度的大小和方向。并且速度是可以积累的，因此我们还具有动量，当力（梯度）改变时就会有一段逐渐加速或逐渐减速的过程。想必你也知道我们会做什么了，通过引入动量的概念，我们可以加速学习的过程，可以在鞍点处继续前行，也可以逃离一些较小的局部最优区域。

接下来，我们稍微正式地来定义**动量**（Momentum）[10]学习算法。首先类似于物理学，我们用变量 v 表示速度，表明参数在参数空间移动的方向及速率，而代价函数的负梯度表示参数在参数空间移动的力。根据牛顿运动定律，动量等于质量乘以速度，而在动量学习算法中，我们假设质量为单位 1，因此速度 v 就可以直接当作动量。我们同时也引入超参数 β，其取值在[0,1]范围之间，用于调节先前梯度（力）的衰减效果。其更新方式如式（5.4）和式（5.5）所示。

$$v=\beta v-\alpha\nabla w \tag{5.4}$$

$$w=w+v \tag{5.5}$$

结合算法 5.1 中的随机梯度算法，可以完成加入动量后改进的随机梯度下降算法，如算法 5.3 所示。

算法 5.3：动量随机梯度下降算法

初始化：
给定数据集 $X=\{x^{(1)},x^{(2)},\cdots,x^{(n)}\}$，数据集标记 $Y=\{y^{(1)},y^{(2)},\cdots,y^{(n)}\}$，初始速度 v，随机采样数据大小 m，训练周期 k，学习器 $f(x;w)$，初始学习率 α，初始动量参数 β。
训练：

For i <= k
{ 1.随机采样 m 条数据：$\{(x^{(1)},y^{(1)})\cdots(x^{(m)},y^{(m)})\}$ 2.计算采样数据平均梯度：$\nabla w=\frac{1}{m}\sum_{j=1}^{m}\frac{\partial L(y^{(j)},f(x^{(j)};w))}{\partial w}$ 3.更新速度：$v=\beta v-\alpha\nabla w$ 4.更新参数：$w=w+v$ }

在随机梯度下降中，每一步走多远是简单的梯度乘以学习率，而在动量学习算法中，每一步走多远依赖的是过去的速度以及当前的力（梯度）。速度 v 用于累加各轮训练的参数梯度，β 越大先前梯度对于本轮训练梯度的影响就越大。假设每轮训练的梯度方向都是相同的，就如同小球从斜坡一直往下滚落，但由于衰减因子 β 的存在，小球并不会一直加速往下，而是达到速度的最大值后就匀速前行。我们假设每轮获得的梯度都是相同的，那该速度的最大值就如式（5.6）所示。

$$v_{\max}=\frac{\alpha\|\nabla w\|}{1-\beta} \tag{5.6}$$

也可以将这一过程理解为受到空气阻力的加速运动，由于速度越大，空气阻力就越大。因此一个受力运动的小球最终会加速到一个受力平衡的状态，然后以最大速度运动。如果 $\beta=0.9$，那么其最大速度就相当于梯度下降的 10 倍。

在实践中，常用的 β 取值可以是 0.5、0.9 或者是 0.99。当然也可以像学习率 α 的调整一样来根据训练轮数的增加自适应地衰减，但对于 β 的调整，并没有 α 的调整重要，因此，不太需要作为超参数进行选择，正常情况下，取值适当即可。

5.4 AdaGrad 和 RMSProp

在前面的小节中，我们都使用一个全局的学习率，所有的参数都是统一步伐整齐向前，但这种“蛮横”的行为是有些问题的。我们将深度学习优化的过程想象成在一个复杂多变的深山中找出路，群山之中有低谷、鞍点、悬崖和高原，如果我们真的只是看一步走一步（梯度下降），或者奔跑向前（动量学习），那我们可能会摔得“头破血流”，接下来我们就个性化的针对**每一个参数单独地配置学习率**。

- AdaGrad

AdaGrad[11]算法其实很简单，就是将每一维各自的历史梯度的平方叠加起来，然后在更新的时候除以该历史梯度值即可。例如，针对第 i 参数，算法如下。

首先，如式（5.7）所示，我们将当前梯度的平方累加在 cache 中，使用平方的原因是去除梯度的符号，我们只对梯度的量进行累加。

$$\text{cache}_i=\text{cache}_i+(\nabla w_i)^2 \tag{5.7}$$

其次，如式（5.8）所示，在更新参数时，学习率需要除以根号 cache，其中 $\delta=10^{-7}$，防止数值溢出。

$$w_i = w_i - \frac{\alpha}{\sqrt{\text{cache}_i} + \delta} \nabla w_i \quad (5.8)$$

从式（5.8）中可以看出，AdaGrad 使得参数在累积的梯度量较小时(<1)，放大学习率，使网络的训练更加快速。在梯度的累积量较大时(>1)，缩小学习率，延缓网络训练。这就好比网络训练开始时要勇往直前，当走完一段距离后要小心翼翼。那么 AdaGrad 有什么问题吗？

细心的读者可能早已观察出来了，AdaGrad 很容易受到“过去”的影响，因为梯度很容易就会累积到比较大的值，此时学习率就会被降低得非常厉害。因此 AdaGrad 很容易**过分降低学习率**。那么要如何改进该算法呢？那就很简单了，做一只快乐的小金鱼，忘记“过去”就好。

- RMSProp

虽然 AdaGrad 理论上有些比较好的性质，但在实践中，优化神经网络却十分不友好。其原因就在于随着训练周期的增长，学习率降低得很快。

因此，RMSProp[12]算法就在 AdaGrad 基础上引入衰减因子，如式（5.9）所示，RMSProp 算法在进行梯度累积的时候，会对“过去”与“现在”做一个权衡。通过超参数 β 来调节衰减量，常用的取值有 0.9 或 0.5。

$$\text{cache}_i = \beta \cdot \text{cache}_i + (1-\beta)(\nabla w_i)^2 \quad (5.9)$$

在参数更新阶段，和 AdaGrad 相同，如式（5.10）所示，学习率除以历史梯度总和即可。

$$w_i = w_i - \frac{\alpha}{\sqrt{\text{cache}_i} + \delta} \nabla w_i \quad (5.10)$$

在实践中，RMSProp 更新方式对于深度学习网络十分的高效，是深度学习中最有效的更新方式之一。

5.5 Adam

Adam[13]的名称来源于“adaptive moments”，可以将其看作为 Momentum + RMSProp 的**微调**版本。该方法是目前深度学习中最流行的优化方法，在默认情况下，我们都推荐使用 Adam 作为参数更新方式。

首先，如式（5.11）所示，计算当前最小批量数据梯度 g。

$$g = \frac{1}{m} \sum_{j=1}^{m} \frac{\partial L(y^{(j)}, f(x^{(j)}; w))}{\partial w} \quad (5.11)$$

然后，如式（5.12）所示，类似于动量学习法，计算衰减梯度 v。

$$v = \beta_1 \cdot v + (1-\beta_1) g \quad (5.12)$$

然后，如式（5.13）所示，类似于 RMSProp 学习法，计算衰减学习率 r。

$$r = \beta_2 \cdot r + (1-\beta_2) g^2 \quad (5.13)$$

最后，如式（5.14）所示，更新参数。

$$w = w - \frac{\alpha}{\sqrt{r} + \delta} v \tag{5.14}$$

以上就是 Adam 算法，是不是很简单。但还有一点小问题，那就是在开始时梯度会非常小，r 和 v 经常会接近于 0，因此我们还需要做一步“热身”工作。如式（5.15）所示，我们将 r 和 v 分别除以 1 减去各自衰减率的 t 次方之差。

$$vb = \frac{v}{1-\beta_1^t}, \quad rb = \frac{r}{1-\beta_2^t} \tag{5.15}$$

其中 t 表示训练的次数，因此仅仅在训练的前几轮中根据衰减因子来放大各自值，很快 vb 和 rb 就会退化为 v 和 r。Adam 学习算法如下所示。

算法 5.4：Adam 学习算法
初始化： 给定数据集 $X=\{x^{(1)},x^{(2)},\cdots,x^{(n)}\}$，数据集标记 $Y=\{y^{(1)},y^{(2)},\cdots,y^{(n)}\}$，初始速度 v，随机采样大小 m，训练周期 k，学习器 $f(x;w)$，初始学习率 α，动量衰减参数 β_1，学习率衰减参数 β_2，$\delta=10^{-7}$。 训练： For t < k {　1.随机采样 m 条数据：$\{(x^{(1)},y^{(1)})\cdots(x^{(m)},y^{(m)})\}$ 　2.计算当前采样数据梯度：$g=\frac{1}{m}\sum_{j=1}^{m}\frac{\partial L(y^{(j)},f(x^{(j)};w))}{\partial w}$ 　3.更新当前速度：$v=\beta_1\cdot v+(1-\beta_1)g$ 　4.更新当前学习率：$r=\beta_2\cdot r+(1-\beta_2)g^2$ 　5.更新训练次数：$t=t+1$ $$vb=\frac{v}{1-\beta_1^t},\quad rb=\frac{r}{1-\beta_2^t}$$ 　6.更新参数：$w=w-\frac{\alpha}{\sqrt{rb}+\delta}vb$　}

5.6　参数初始化策略

深度学习的优化过程可以看作是下山的过程，山路崎岖，有鞍点，有局部最优点，也有悬崖点。之前我们学习了很多下山的方法，如 SGD、Momentum、RMSProp 和 Adam 都是不错的方法。但无论使用何种学习方式，都避免不了同一个问题，那就是下山的起始点问题，也就是**参数的初始化问题**。

如果参数初始点不好，学习算法可能根本无法收敛，也可能造成学习算法不稳定，容易遭遇数值溢出而无法学习。初始化的参数即使是可收敛的，不同的参数也决定着学习的收敛速度以及能否降低到足够低的损失值区域。初始化不仅影响训练效果，还能影响模型的泛化性能，这也许就是人们所说的输在起跑线上吧！

现代的参数初始化策略通常是简单并且带有试探性的，就目前而言，神经网络的优化策

略仍然没有充分的完善，想要设计优秀的初始化策略是一件非常困难的工作。大多数初始化策略都是基于已实现网络的某些优良属性，但我们并不知道这些属性应该应用在何种环境中。

目前，我们能够确定的一种性质是在不同的神经元中，应该尽量避免神经元参数出现对称情况。如果两个隐藏单元使用相同的激活函数，并且连接到相同的输入，那么每个神经元必须要有不同参数。因为参数相同，在确定的学习算法下，其参数的更新就会相同，这就会出现神经元冗余的情况。

在初始化权重参数时，我们几乎都采用均匀分布或高斯分布的随机初始化方式。选择均匀分布或者高斯分布没有好坏之分，但分布函数的**取值范围**（标准差）对于优化过程以及泛化能力却有着巨大的影响。

较大的初始化权重范围可以有效地避免参数对称线性，减少神经元冗余现象，也可以帮助**减轻训练网络时的梯度消失问题**。但太大的初始化权重也会**导致梯度爆炸的问题**，同时在一些使用诸如 Sigmoid 激活函数那样的神经元中，也容易出现**过饱和现象**，致使修改神经元梯度几乎为零。

从正则化以及最优化的角度出发，我们可能会得出完全不同的参数初始化策略。以最优化的观点来看，权重应该足够大，这样才能较好地传递信息；但从正则化的角度出发，我们却希望参数越小越好。由于神经网络中存在局部最优和鞍点等困难，最终训练后的神经网络权重会**非常靠近初始化时的权重**。综合正则化要求以及训练权重接近初始化权重的事实，我们实际上是在权重中加入了一条参数**均值为 0** 的高斯分布先验知识。而参数尽可能小这一先验知识，其实也就是所谓的**稀疏性原则**，即神经元不应该连接到全部神经元中，而是应该连接到需要连接的神经元中。

通常情况下，在 m 输入 n 输出的单层神经元中，我们可以使用均匀分布将权重进行随机采样初始化，如式（5.16）所示。

$$W_{i,j} \sim U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right) \tag{5.16}$$

也可以如式（5.17）所示，使用**标准初始化**（Normalized Initialization）[14]。

$$W_{i,j} \sim U\left(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}\right) \tag{5.17}$$

需要注意的是，该式是假设神经网络仅仅进行一组矩阵链式相乘，并不进行非线性转化，但实际的神经网络肯定是违背了这种假设的。由于很多针对线性模型设计的策略，也在对应的非线性模型中表现良好，因此，这种方法也被广泛应用在非线性神经元中。

在具体实践中，我们通常要将参数的取值范围设置为可缩放的因子，作为一种超参数通过验证集来调整。上述初始化的一个缺点在于所有初始化权重都会具有相同的标准差，当每层的神经元数目太多时，神经元的权重就会变得极其的小，解决的方式是一种被称为**稀疏初始化**（Sparse Initialization）[15]的策略，该方法强制每个神经元中要有 k 个非零的权重。稀疏初始化可以在神经元之间实现更强的多样性，但也施加了一个非常强的先验在权重中。假设权重较大的神经元恰好是需要修正的神经元，那可能需要花费很长的时间来缩小误差值。如果计算资源允许，将每层的权重的标准差都设置成一个超参数是不错的选择。

上述的初始化策略，其实都是随机的初始化参数，需要我们仔细地选择参数。人工智能的一大目标就是尽可能地减少人为参与，那么将参数的初始化当作是机器学习任务去学习又

如何呢？其中非常著名的策略就是使用非监督学习方法逐层地学习深度模型的参数[16]，将其作为监督学习模型的参数初始化过程，这是深度学习中非常重要的方法，也是深度学习研究的主要方向之一。

5.7　批量归一化

批量归一化（Batch Normalization）[17]并不能算作是一种最优化算法，但其却是近年来优化深度神经网络最有用的技巧之一，并且这种方法非常的简洁方便，可以和其他算法兼容使用，大大加快了深度模型的训练时间。

5.7.1　BN 算法详解

那什么是批量归一化呢？首先，归一化就是将数据的输入值减去其均值然后除以数据的标准差，几乎所有数据预处理都会使用这一步骤。而深度学习也可以认为是逐层特征提取的过程，那每一层的输出其实都可以理解为经过特征提取后的数据。因此，批量归一化方法的“归一化”所做的其实就是在网络的每一层都进行数据归一化处理，但每一层对所有数据都进行归一化处理的计算开销太大，因此就和使用最小批量梯度下降一样，批量归一化中的“批量”其实是采样一小批数据，然后对该批数据在网络各层的输出进行归一化处理。

假设我们一次采样 m 条数据训练，用 $H_{i,j}^{(k)}$ 表示训练第 k 条数据时，第 j 层的第 i 神经元的输出值；$\mu_{i,j}$ 表示这批数据在第 j 层的第 i 神经元处的**平均**输出值；$\sigma_{i,j}$ 表示这批数据在第 j 层的第 i 神经元处输出值的**标准差**。批量归一化后的输出值就如式（5.18）所示。

$$H_{i,j}^{\prime(k)}=\frac{H_{i,j}^{(k)}-\mu_{i,j}}{\sigma_{i,j}} \tag{5.18}$$

其中神经元输出的**均值** $\mu_{i,j}$ 如式（5.19）所示。

$$\mu_{i,j}=\frac{1}{m}\sum_{k=1}^{m}H_{i,j}^{(k)} \tag{5.19}$$

神经元输出值的**标准差** $\sigma_{i,j}$ 如式（5.20）所示。

$$\sigma_{i,j}=\sqrt{\delta+\frac{1}{m}\sum_{k=1}^{m}(H_{i,j}^{(k)}-\mu_{i,j})^2} \tag{5.20}$$

其中 δ 是一个很小的常数，目的是为了防止 $\sqrt{0}$ 的产生。批量归一化的目的其实很简单，就是把神经网络每一层的输入数据都调整到**均值为零，方差为 1 的标准正态分布。**那为什么这样做呢？

这还要从深度神经网络最大的梦魇——梯度消失讲起。假设我们使用 Sigmoid 作为神经元的激活函数，当输出值较大时，Sigmoid 函数就会进入饱和区域，导致其导数几乎为零，即使我们知道需要矫正该神经元，也会因为梯度太小而无法训练。如图 5-6 所示，Sigmoid 激活函数在[-2, 2]之间的取值是一段近似线性的区域。想必你也猜到了吧，其实 BN 算法所做的

就是把输入值尽可能地归一化在激活函数这一狭窄区域。

但这又引入了另一个问题，我们知道多层线性神经网络其实可以用一层线性网络来表示，那我们使用 BN 算法将输入值投射到激活函数的线性区域，会不会使得深度网络能力下降呢？答案是肯定的，如果仅仅是归一化各层输入值到一个近似的线性区域，我们的深层网络能力将大大降低。

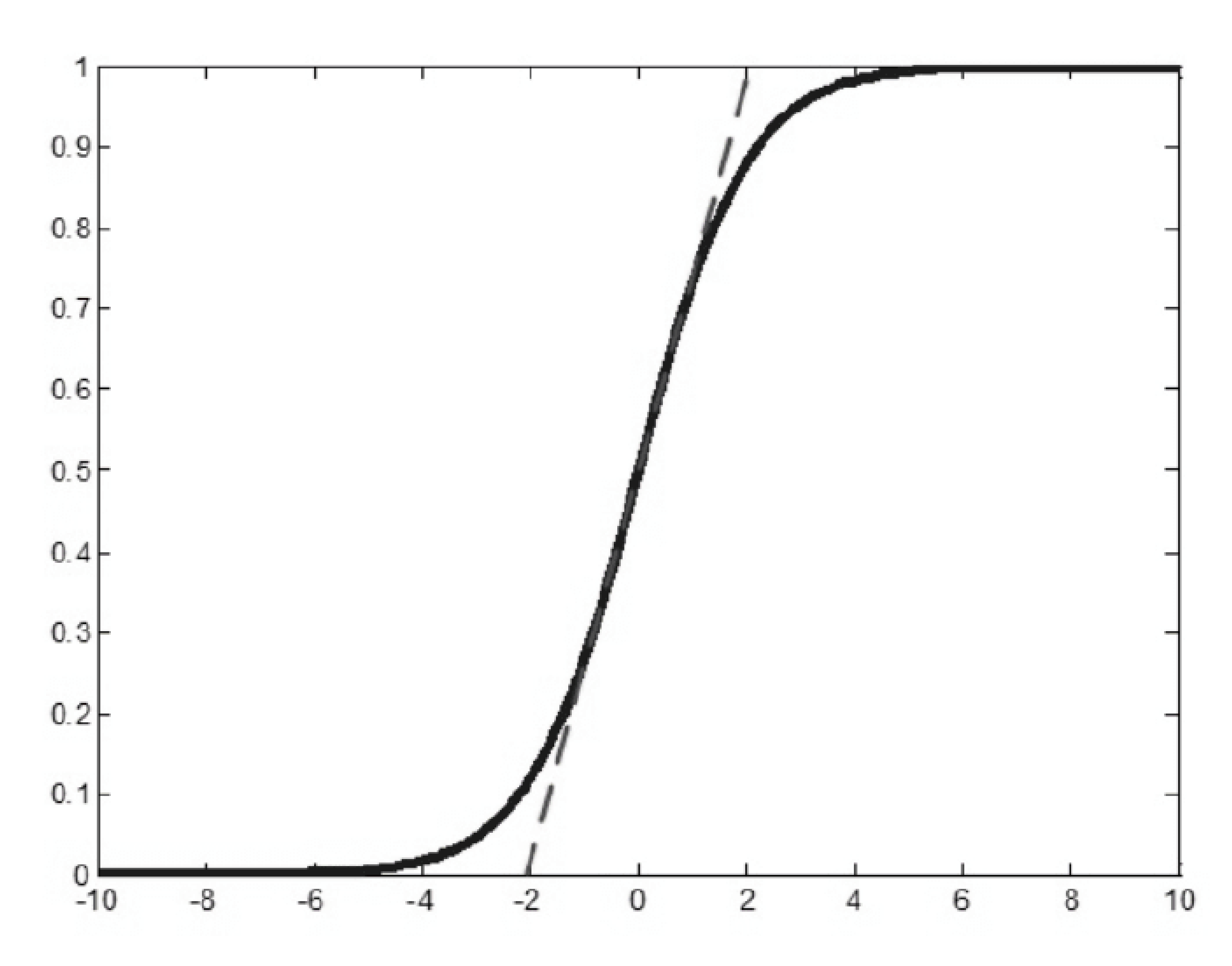


图 5-6　Sigmoid 激活函数与线性函数比较示意图

因此，BN 算法其实还有另一个步骤，那就是再将归一化的数据放大，平移回非线性区域，如式（5.21）所示，我们引入 γ 和 β 两个可学习参数，调整归一化的输出值。

$$X_{i,j}^{(k)} = \gamma_{i,j} H_{i,j}'^{(k)} + \beta_{i,j} \tag{5.21}$$

变量 γ 和 β 是允许新变量有任意均值和标准差的学习参数，这似乎不合理，为什么我们将均值设为 0，然后又引入参数允许它被重设为 β 呢？这是因为新的参数不但可以表示旧参数的非线性能力，而且新参数还可以消除层与层之间的关联，具有相对独立的学习方式。在旧参数中，H 的均值和方差取决于 H 下层中参数的复杂关联。在新参数中，$\gamma \cdot H' + \beta$ 的均值与方差仅仅由本层决定，新参数很容易通过梯度下降来学习。

- 运行时平均（running averages）

由于训练时我们仅仅对批量采样数据进行归一化处理，该批数据的均值和方差不能代表全体数据的均值和方差。因此，我们需要在每批训练数据归一化后，累积其均值和方差。当训练完成后再求出总体数据的均值和方差，然后再在测试时使用，但累积每一次数据的均值和方差太过麻烦，在实践中，我们经常使用运行时的均值和方差代替全体数据的均值和方差。如式（5.22）和式（5.23）所示，和动量学习法类似，我们引入衰减因子对均值和方差进行衰减累积。

$$\mu = \text{decay} \cdot \mu + (1 - \text{decay}) \mu_{\text{sample}} \tag{5.22}$$

$$\sigma = \text{decay} \cdot \sigma + (1 - \text{decay})\sigma_{\text{sample}} \tag{5.23}$$

5.7.2　BN 传播详解

BN 算法的思想虽然非常简单，但计算过程也比较烦琐，我们在编程时可能会感到手足无措。接下来，我们将 BN 算法传播过程拆解成详细的计算流图，希望能在你编码时提供帮助。也可以先跳过本部分内容，当你在 5.8.5 节遇到困难时，再回过头仔细地阅读该节。

如图 5-7 所示，我们将 BN 算法传播过程拆解成 9 个计算步骤，每个步骤仅完成一个计算任务，你需要保存每个计算步骤的中间结果，该结果会在反向传播时使用。

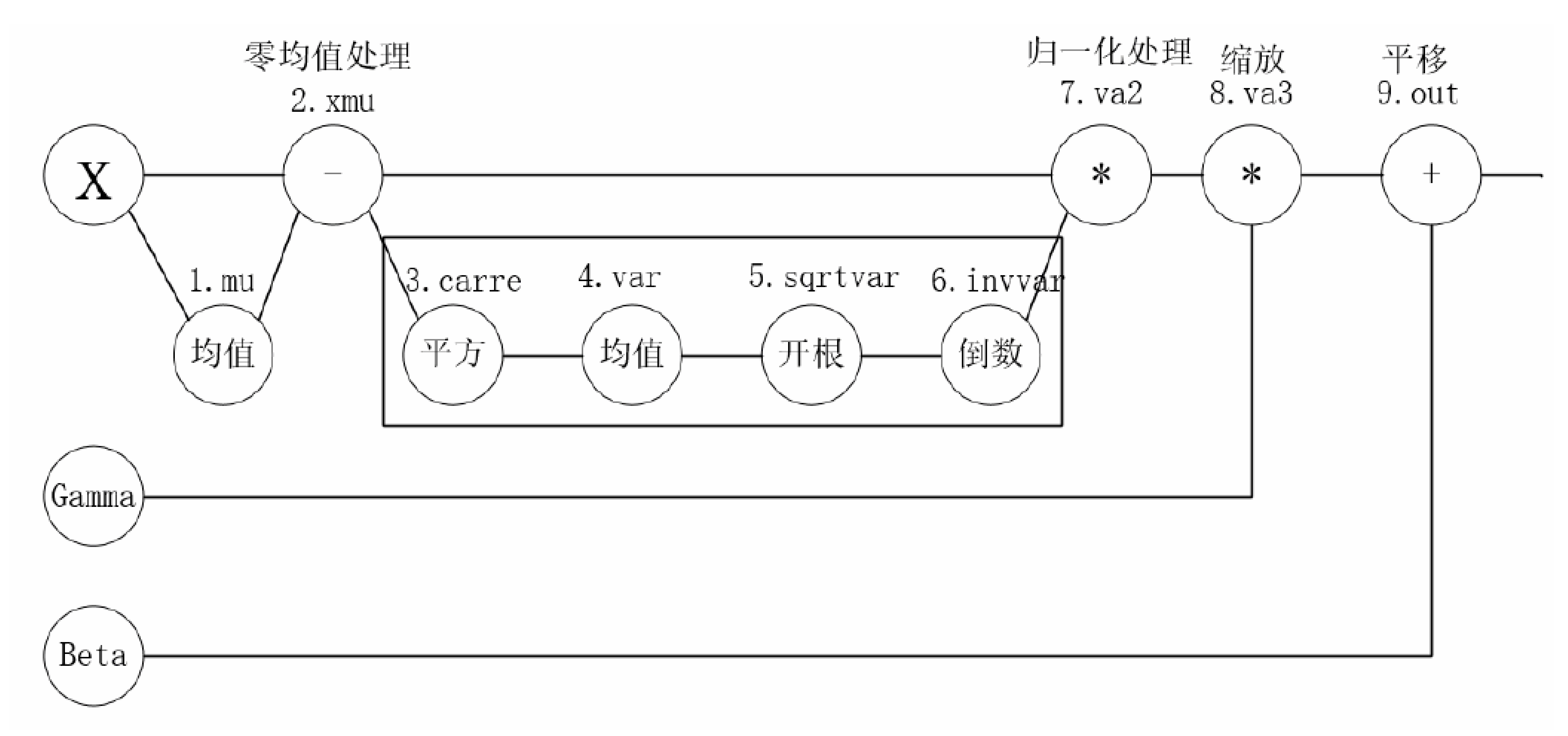


图 5-7　BN 传播示意图

- BN 前向传播

1. 计算采样数据均值：$mu = \frac{1}{m}\sum_{i=1}^{m} x_i$

2. 将数据平移为**零均值**：$xmu_i = x_i - mu$

3. 计算平方项：$carre_i = xmu_i^2$

4. 计算数据方差：$var = \frac{1}{m}\sum_{i=0}^{m} carre_i$

5. 计算数据标准差：$sqrtvar = \sqrt{var + \delta}$

6. 计算数据标准差倒数：$invvar = \frac{1}{sqrtvar}$

7. 采样数据**归一化**：$va2_i = xmu_i \times invvar$

8. 采样数据缩放：$va3_i = gamma \times va2_i$

9. 采样数据平移：$out_i = va3_i + beta$

- BN 反向传播

9. 计算 beta 梯度：$dbeta = \sum_{i=1}^{m} dout_i$

计算 va3 梯度：$dva3_i = dout_i$

8. 计算 va2 梯度：$dva2_i = gamma \times dva3_i$

计算 gamma 梯度：$dgamma = \sum_{i=1}^{m} va2_i \times dva3_i$

7. 计算 xmu1 梯度：$dxmu1_i = invvar \times dva2_i$

计算 invvar 梯度：$dinvvar = \sum_{i=1}^{m} xmu_i \times dva2_i$

6. 计算 sqrtvar 梯度：$dsqrtvar = \frac{-1}{sqrtvar^2} \times dinvvar$

5. 计算 var 梯度：$dvar = 0.5 \times (var + \delta)^{-0.5} \times dsqrtvar$

4. 计算 carre 梯度：$dcarre_i = \frac{1}{m} dvar$

3. 计算 xmu2 梯度：$dxmu2_i = 2 \times xmu_i \times dcarre_i$

计算 xmu 梯度：$dxmu_i = dxmu1_i + dxmu2_i$

2. 计算 x1 梯度：$dx1_i = dxmu_i$

计算 mu 梯度：$dmu = -\sum_{i=1}^{m} dxmu_i$

1. 计算 x 梯度：$dx_i = \frac{dmu_i}{m} + dx1_i$

5.8 深度学习编码实战下

在本章节的练习中，首先我们要完成 Momentum、RMSProp 和 Adam 三种优化方法的代码编写。在此之后，我们将重点进行 BN 算法的前向传播，反向传播的实现。最后，我们测试 BN 算法的性能，说明它是如何降低参数初始化时标准差的影响来加快网络的训练。本章

我们将逐步完成以下操作。

- 编码实现 Momentum 算法;
- 编码实现 RMSProp 算法;
- 编码实现 Adam 算法;
- 编码实现 BN 前向传播;
- 编码实现 BN 反向传播;
- 编码实现 BN 全连接网络。

接下来，打开“第 5 章练习-神经网络下.ipynb”文件，进入本章练习。首先是我们已经非常熟悉的库文件导入代码模块，本章的练习文件全部存放在“DLAction/ classifiers /classifiers/chapter5”目录下。

导入库文件与数据：

```
# -*- coding: utf-8 -*-
import time
import numpy as np
import matplotlib.pyplot as plt
from classifiers.chapter5 import *
from utils import *
%matplotlib inline
plt.rcParams[ 'figure.figsize' ] = ( 10.0, 8.0 )
plt.rcParams[ 'image.interpolation' ] = 'nearest'
plt.rcParams[ 'image.cmap' ] = 'gray'
%load_ext autoreload
%autoreload 2
def rel_error (x, y ) :
  """ 返回相对误差 """
  return np.max( np.abs( x – y ) / ( np.maximum( 1e-8,   np.abs( x ) + np.abs( y ) ) ) )
data = get_CIFAR10_data( )
for k, v in data.iteritems( ) :
  print '%s: ' % k, v.shape
```

5.8.1　Momentum

接下来，打开“DLAction/classifiers/chapter5/updater.py”文件，完成 sgd_momentum 函数的编码工作。我们使用 momentum 参数作为动量衰减因子，完成速度的计算后，需要将其保存在 config['velocity']中。

动量更新函数代码模块：

```
def sgd_momentum( w, dw, config = None ) :
```

```
"""
动量随机梯度下降更新规则。
config 使用格式：
- learning_rate: 学习率。
- momentum: [0,1]的动量衰减因子，0 表示不使用动量，即退化为 SGD。
- velocity: 和 w，dw 形状相同的速度。
"""
if config is None: config = { }
config.setdefault( 'learning_rate', 1e-2 )
config.setdefault( 'momentum', 0.9 )
v = config.setdefault( 'velocity', np.zeros_like( w ) )
next_w = None
####################################################################
#                    任务：实现动量更新。                              #
#     更新后的速度存放在 v 中，更新后的权重存放在 next_w 中。            #
####################################################################


####################################################################
#                          结束编码                                  #
####################################################################
config[ 'velocity' ] = v
return next_w, config
```

实现上述代码块后，运行下列代码进行检验，相对误差应该小于 1e-8。

```
动量更新规则代码检验：
N, D = 4, 5
w = np.linspace( -0.4, 0.6, num = N * D ).reshape( N, D )
dw = np.linspace( -0.6, 0.4, num = N * D ).reshape( N, D )
v = np.linspace( 0.6, 0.9, num = N * D ).reshape( N, D )
config = { 'learning_rate': 1e-3, 'velocity': v }
next_w, _ = sgd_momentum( w, dw, config = config )
expected_next_w = np.asarray( [
  [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789 ],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526 ],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263 ],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096      ] ] )
expected_velocity = np.asarray( [
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158  ],
```

```
  [ 0.61138947,   0.62554737,   0.63970526,   0.65386316,   0.66802105 ],
  [ 0.68217895,   0.69633684,   0.71049474,   0.72465263,   0.73881053 ],
  [ 0.75296842,   0.76712632,   0.78128421,   0.79544211,   0.8096       ] ] )
print '更新权重误差: ', rel_error( next_w, expected_next_w )
print '速度误差: ', rel_error( expected_velocity, config[ 'velocity' ] )
```

momentum 正确编码后的检验结果：

```
更新权重误差:  8.88234703351e-09
速度误差:  4.26928774328e-09
```

当实现 momentum 更新规则后，我们使用 5 层隐藏层神经网络进行测试，正常情况下，加入动量方法后会比原始的 SGD 算法收敛得更快一些。这两种算法损失函数、训练精度和验证精度比较示意图分别如图 5-8、图 5-9 和图 5-10 所示。

Momentum 与 SGD 算法比较代码块：

```
num_train = 4000
small_data = {
  'X_train': data[ 'X_train' ] [ : num_train ],
  'y_train': data[ 'y_train' ] [ : num_train ],
  'X_val': data[ 'X_val' ],
  'y_val': data[ 'y_val' ],
}
trainers = { }
for update_rule in [ 'sgd', 'sgd_momentum' ]:
  model = FullyConnectedNet( hidden_dims =[ 100, 100, 100, 100, 100 ],
 weight_scale = 7e-2 )
  trainer = Trainer( model, small_data,
                   num_epochs = 10, batch_size = 100,
                   update_rule = update_rule,
                   updater_config = { 'learning_rate': 1e-3, },
                   verbose = False )
  trainers[ update_rule ] = trainer
  trainer.train( )
plt.subplot( 3, 1, 1 )
plt.title( 'Training loss', fontsize = 18 )
plt.xlabel( 'Iteration', fontsize = 18 )
plt.ylabel( 'Loss', fontsize = 18 )
plt.subplot( 3, 1, 2 )
plt.title( 'Training accuracy', fontsize = 18 )
plt.xlabel( 'Epoch', fontsize = 18 )
```

```
plt.ylabel( 'Accuracy', fontsize = 18 )
plt.subplot( 3, 1, 3 )
plt.title( 'Validation accuracy', fontsize = 18 )
plt.xlabel( 'Epoch', fontsize = 18 )
plt.ylabel( 'Accuracy', fontsize = 18 )
plt.subplots_adjust( left = 0.08, right = 0.95, wspace = 0.25, hspace = 0.25 )
a = { 'sgd': 'o', 'sgd_momentum': '*' }
for update_rule, trainer in trainers.iteritems( ):
    plt.subplot( 3, 1, 1 )
    plt.plot( trainer.loss_history, a[update_rule], label = update_rule )
    plt.subplot( 3, 1, 2 )
    plt.plot( trainer.train_acc_history, '-'+a[ update_rule ], label = update_rule )
    plt.subplot( 3, 1, 3 )
    plt.plot( trainer.val_acc_history, '-'+a[ update_rule ], label = update_rule )
for i in [ 1, 2, 3 ]:
    plt.subplot( 3, 1, i )
    plt.legend( loc = 'upper center', ncol = 4 )
plt.gcf( ).set_size_inches( 15, 15 )
plt.show( )
```

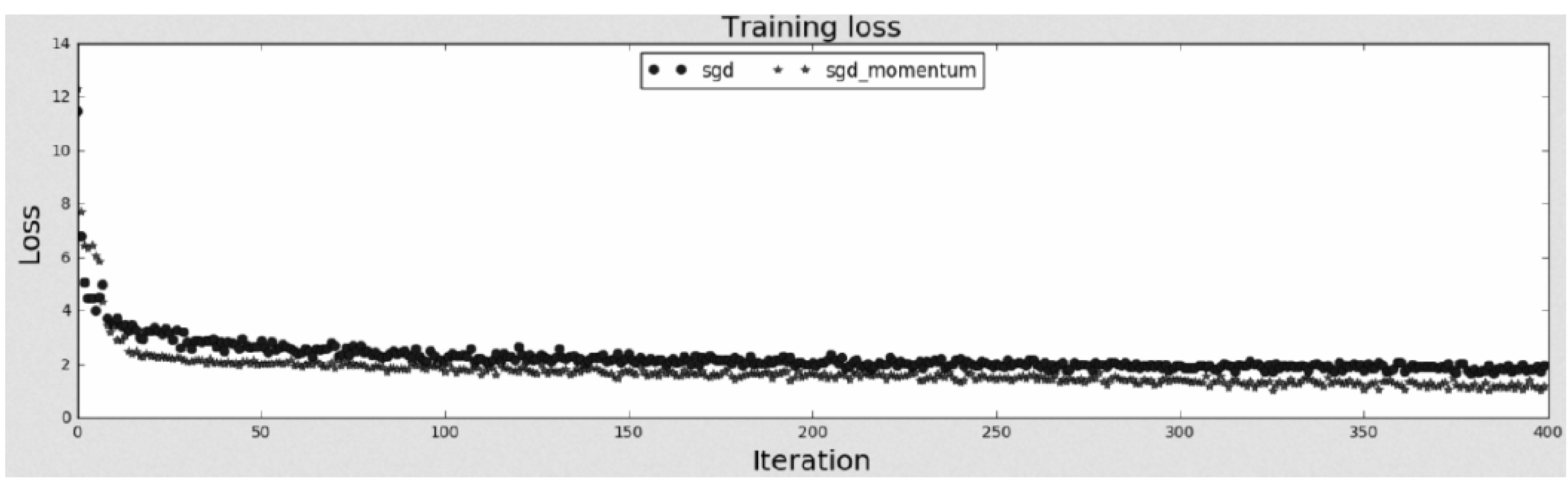


图 5-8　SGD 与 Momentum 损失函数比较示意图

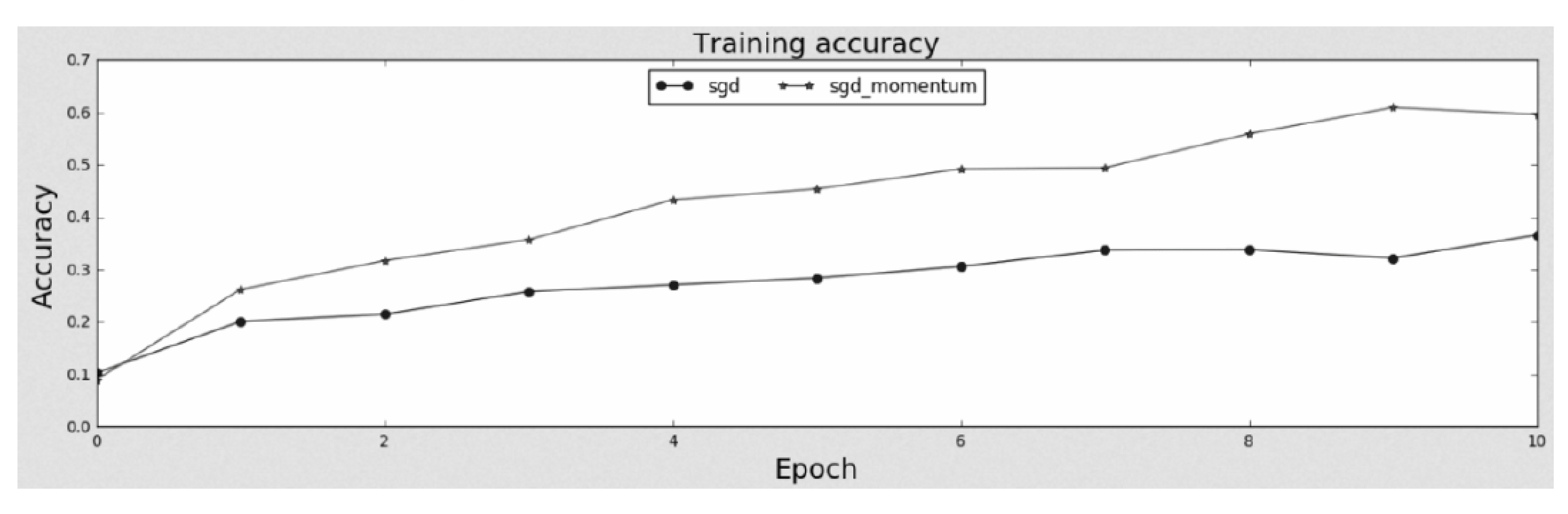


图 5-9　SGD 与 Momentum 训练精度比较示意图

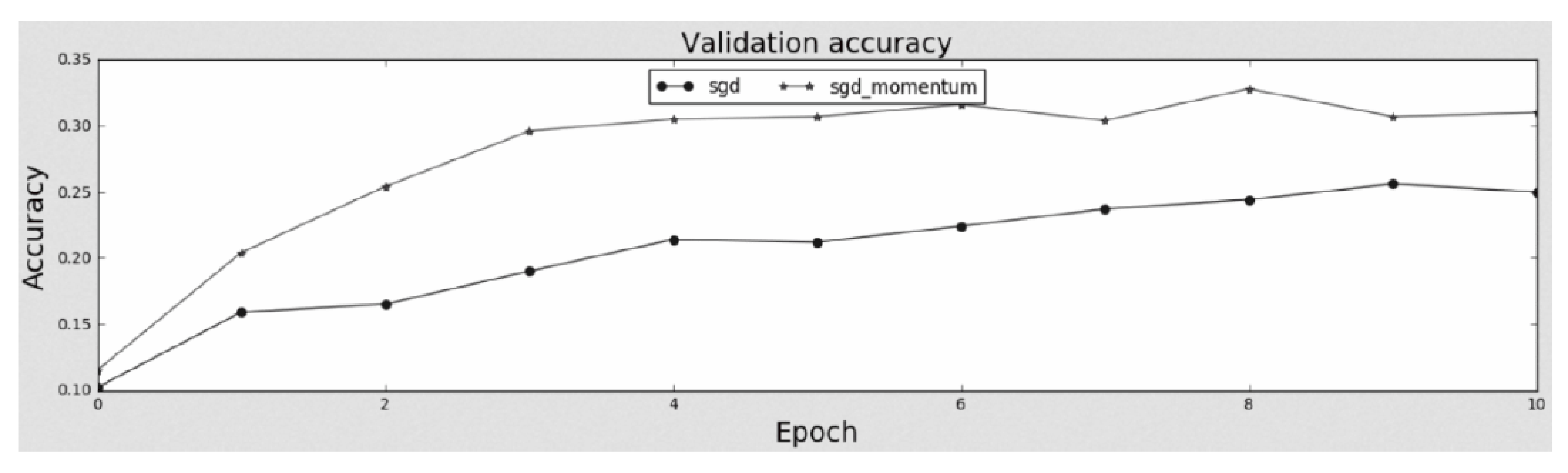


图 5-10　SGD 与 Momentum 验证精度比较示意图

由图 5-8 至图 5-10 可以看出，动量学习法的收敛速度更快，加快了网络的训练效率。因此在使用 SGD 时，加入动量的更新方式是比较好的选择。

5.8.2　RMSProp

接下来我们编码实现 RMSProp 的更新算法。我们使用 config['cache']保存历史积累梯度，使用 config['decay_rate']计算梯度衰减，并且别忘了加入 config['epsilon']防止溢出。

RMSProp 更新规则代码块：

```
def rmsprop(w, dw, config = None ) :
  """
  RMSProp 更新规则。
  config  使用格式：
  - learning_rate:  学习率。
  - decay_rate:历史累积梯度衰减率因子，取值为[ 0, 1 ]。
  - epsilon:  避免溢出的小数。
  - cache:历史梯度缓存。
  """
  if config is None: config = { }
  config.setdefault( 'learning_rate', 1e-2 )
  config.setdefault( 'decay_rate', 0.99 )
  config.setdefault( 'epsilon', 1e-8 )
  config.setdefault( 'cache', np.zeros_like( w ) )
  next_w = None
  ###########################################################################
  #                              任务：实现  RMSProp  更新。                        #
  #  将更新后的权重保存在 next_w 中，将历史梯度累积存放在 config[ 'cache' ]中。  #
  ###########################################################################



  ############################################################################
```

```
#                              结束编码                              #
##########################################################################
return next_w, config
```

完成 RMSProp 算法的编码后，使用下列代码进行检验，实现的相对误差应该小于 1e-7。

RMSProp 更新函数代码检验块：

```
# 测试 RMSProp ;相对误差应该小于 1e-7。
N, D = 4, 5
w = np.linspace( -0.4, 0.6, num = N * D ).reshape( N, D )
dw = np.linspace( -0.6, 0.4, num = N * D ).reshape( N, D )
cache = np.linspace( 0.6, 0.9, num = N * D ).reshape( N, D )
config = { 'learning_rate': 1e-2, 'cache': cache }
next_w, _ = rmsprop( w, dw, config = config )
expected_next_w = np.asarray( [
  [ -0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.18467247 ],
  [ -0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.07515774 ],
  [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.33532447 ],
  [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.59576619 ] ] )
expected_cache = np.asarray( [
  [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.65804321 ],
  [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.73484377 ],
  [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.81302936 ],
  [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926     ] ] )
print '权重更新误差: ', rel_error( expected_next_w, next_w )
print 'cache  误差: ', rel_error( expected_cache, config[ 'cache' ] )
```

RMSProp 正确编码后的测试结果：

```
权重更新误差:  9.50264522989e-08
cache  误差:  2.64779558072e-09
```

5.8.3 Adam

接下来我们实现 Adam 更新算法，需要将累积梯度、累积梯度平方，分别保存在 config['m'] 和 config['v']中，还要加入 config['epsilon']项，防止除零。我们实现的是“初始加速”版本，因此还需要保存迭代次数。

Adam 更新函数代码块：

```
def adam(w, dw, config = None ) :
  """
```

```
使用 Adam 更新规则，融合了“热身”更新操作。
config 使用格式：
- learning_rate: 学习率。
- beta1: 动量衰减因子。
- beta2: 学习率衰减因子。
- epsilon: 防除 0 小数。
- m: 梯度。
- v: 梯度平方。
- t: 迭代次数。
"""
if config is None: config = { }
config.setdefault( 'learning_rate', 1e-3 )
config.setdefault( 'beta1', 0.9 )
config.setdefault( 'beta2', 0.999 )
config.setdefault( 'epsilon', 1e-8 )
config.setdefault( 'm', np.zeros_like( w ) )
config.setdefault( 'v', np.zeros_like( w ) )
config.setdefault( 't', 0 )
next_w = None
###########################################################################
#                       任务：实现 Adam 更新。                              #
#   将更新后的权重存放在 next_w 中，记得将 m、v 和 t 存放在相应的 config 中。   #
###########################################################################




###########################################################################
#                            结束编码                                      #
###########################################################################
return next_w, config
```

完成 Adam 函数的编码后，我们使用下列代码块进行验证。

Adam 函数代码检验块：

```
N, D = 4, 5
w = np.linspace( -0.4, 0.6, num = N * D ).reshape( N, D )
dw = np.linspace( -0.6, 0.4, num = N * D ).reshape( N, D )
m = np.linspace( 0.6, 0.9, num = N * D ).reshape( N, D )
```

```
v = np.linspace( 0.7, 0.5, num = N * D ).reshape( N, D )
config = { 'learning_rate': 1e-2, 'm': m, 'v': v, 't': 5 }
next_w, _ = adam( w, dw, config = config )
expected_next_w = np.asarray( [
  [ -0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977 ],
  [ -0.1380274,   -0.08544591, -0.03286534,   0.01971428,   0.0722929 ],
  [ 0.1248705,    0.17744702,   0.23002243,   0.28259667,   0.33516969 ],
  [ 0.38774145,   0.44031188,   0.49288093,   0.54544852,   0.59801459 ] ] )
expected_v = np.asarray( [
  [ 0.69966,      0.68908382,   0.67851319,   0.66794809,   0.65738853, ],
  [ 0.64683452,   0.63628604,   0.6257431,    0.61520571,   0.60467385, ],
  [ 0.59414753,   0.58362676,   0.57311152,   0.56260183,   0.55209767, ],
  [ 0.54159906,   0.53110598,   0.52061845,   0.51013645,   0.49966,    ] ] )
expected_m = np.asarray( [
  [ 0.48,         0.49947368,   0.51894737,   0.53842105,   0.55789474 ],
  [ 0.57736842,   0.59684211,   0.61631579,   0.63578947,   0.65526316 ],
  [ 0.67473684,   0.69421053,   0.71368421,   0.73315789,   0.75263158 ],
  [ 0.77210526,   0.79157895,   0.81105263,   0.83052632,   0.85        ] ] )
print '权重更新误差: ', rel_error( expected_next_w, next_w )
print 'v  误差: ', rel_error( expected_v, config[ 'v' ] )
print 'm  误差: ', rel_error( expected_m, config[ 'm' ] )
```

正确编码 Adam 算法的检验结果：

```
权重更新误差:   1.13956917985e-07
v  误差:   4.20831403811e-09
m  误差:   4.21496319311e-09
```

5.8.4 更新规则比较

接下来，我们比较测试 SGD、Momentum、RMSProp 及 Adam。由于 SGD 和 Momentum 已经在 trainers 字典中了，我们只需要实例化 RMSProp 及 Adam 网络即可。但在此之前，请确保已经运行了 SGD 与 Momentum 的比较代码块，图 5-11、图 5-12 与图 5-13 分别为这 4 种算法的损失函数、训练精度与验证精度比较示意图。

SGD，momentum，RMSProp，Adam 比较代码块：

```
learning_rates = { 'rmsprop': 1e-4, 'adam': 1e-3 }
for update_rule in [ 'adam', 'rmsprop' ]:
  model = FullyConnectedNet( hidden_dims = [ 100, 100, 100, 100, 100 ],weight_scale = 7e-2 )
  trainer = Trainer( model, small_data,
```

```
                    num_epochs = 10, batch_size = 100,
                update_rule = update_rule,
                updater_config = {
                  'learning_rate': learning_rates[ update_rule ]
                },
                verbose = False )
  trainers[ update_rule ] = trainer
  trainer.train( )
plt.subplot( 3, 1, 1 )
plt.title( 'Training loss', fontsize = 18 )
plt.xlabel( 'Iteration', fontsize = 18 )
plt.ylabel( 'Loss',fontsize = 18 )
plt.subplot( 3, 1, 2 )
plt.title( 'Training accuracy', fontsize = 18 )
plt.xlabel( 'Epoch', fontsize = 18 )
plt.ylabel( 'Accuracy', fontsize = 18 )
plt.subplot( 3, 1, 3 )
plt.title( 'Validation accuracy', fontsize = 18 )
plt.xlabel( 'Epoch', fontsize = 18 )
plt.ylabel( 'Accuracy', fontsize = 18 )
plt.subplots_adjust( left = 0.08, right = 0.95, wspace = 0.25, hspace = 0.25 )
a[ 'adam' ] = 'D'
a[ 'rmsprop' ] = 'v'
for update_rule, trainer in trainers.iteritems( ):
  plt.subplot( 3, 1, 1 )
  plt.plot( trainer.loss_history, a[ update_rule ], label = update_rule )
  plt.subplot( 3, 1, 2 )
  plt.plot( trainer.train_acc_history, '-'+a[ update_rule ], label = update_rule )
  plt.subplot(3, 1, 3)
  plt.plot( trainer.val_acc_history, '-'+a[ update_rule ], label = update_rule )
for i in [ 1, 2, 3 ]:
  plt.subplot( 3, 1, i )
  plt.legend( loc = 'upper center', ncol = 4 )
plt.gcf( ).set_size_inches( 15, 15 )
plt.show( )
```

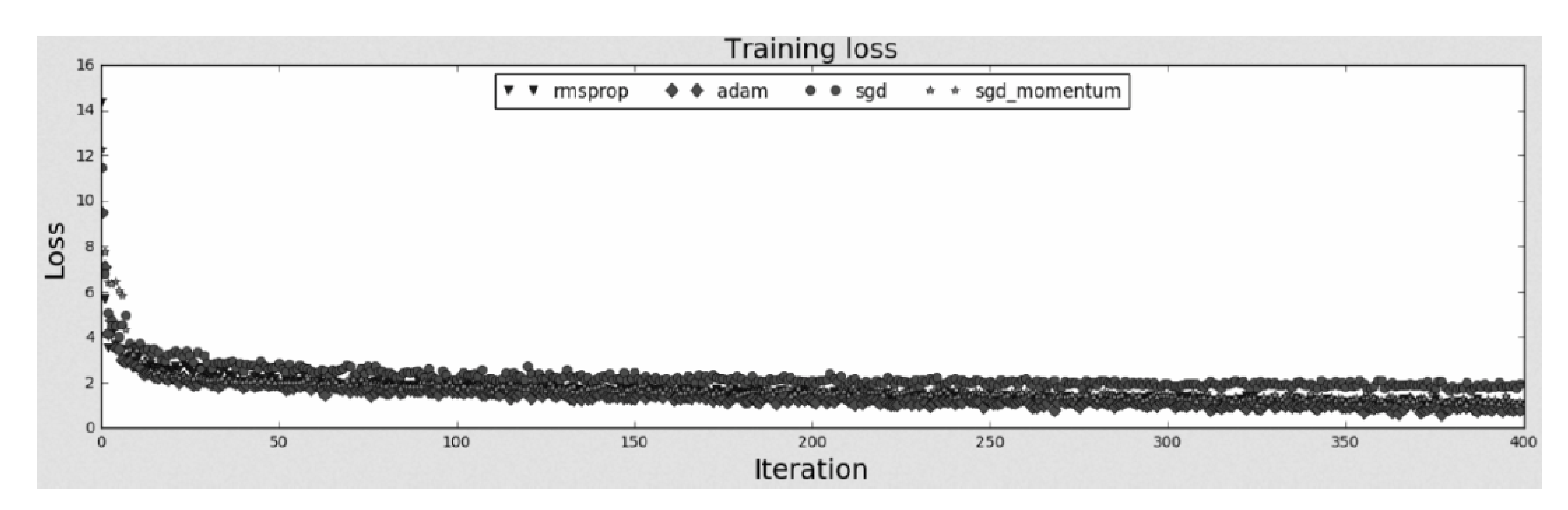


图 5-11　4 种算法损失函数比较示意图

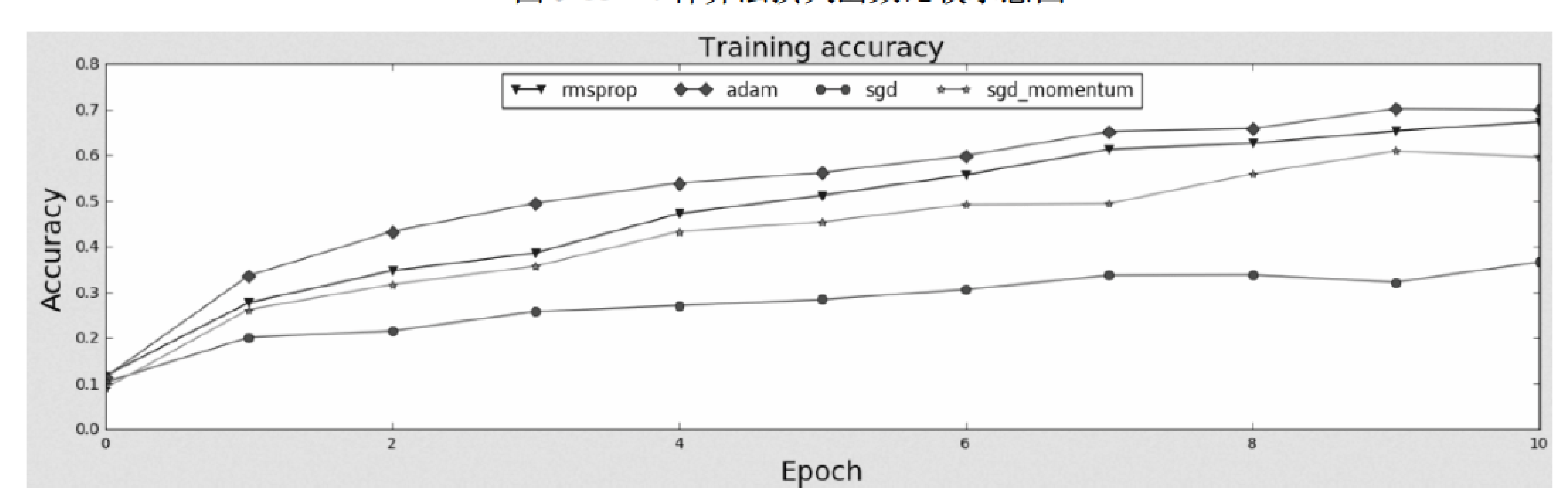


图 5-12　4 种算法训练精度比较示意图

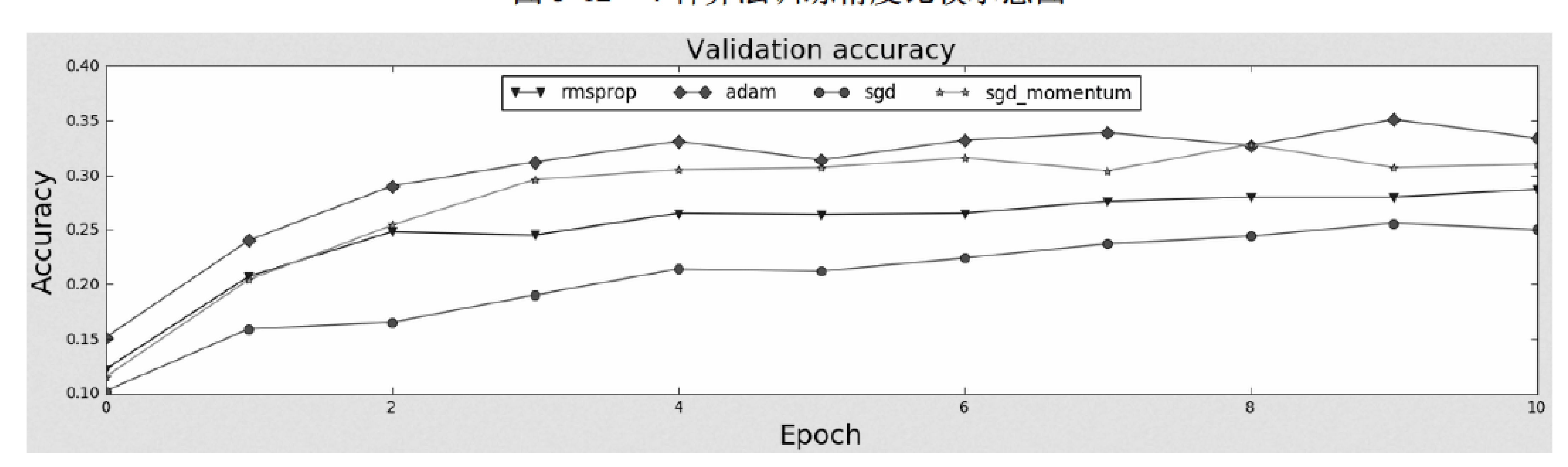


图 5-13　4 种算法验证精度比较示意图

从图 5-11 至图 5-13 可以看出，Adam 算法的收敛效果普遍要比其他三种更新方式要快速高效些。因此在默认情况下，我们都比较推荐使用 Adam 算法进行网络优化。

5.8.5　BN 前向传播

由于该部分内容公式比较多，稍有不慎就会弄晕自己。如果你没有任何思路下手，可以翻看 5.7.2 节中关于 BN 算法详细的推导过程，我们将 BN 算法拆解为 9 个子计算步骤，你只需要依次编码即可。接下来，打开“DLAction/classifiers /chapter5/bn_layers.py”文件，实现 batchnorm_forward 函数，执行 BN 算法的前向传播过程。

batchnorm_forward 函数代码块：
def batchnorm_forward (x, gamma, beta, bn_param) :

```
"""
使用类似动量衰减的运行时平均，计算总体均值与方差 例如：
running_mean = momentum * running_mean + ( 1 – momentum ) * sample_mean
running_var = momentum * running_var + ( 1 – momentum ) * sample_var
Input:
- x: 数据( N, D )。
- gamma: 缩放参数( D, )。
- beta: 平移参数( D, )。
- bn_param: 字典型，使用下列键值：
  - mode: 'train' 或 'test'。
  - eps: 保证数值稳定。
  - momentum: 运行时平均衰减因子。
  - running_mean: 形状为( D, )的运行时均值。
  - running_var : 形状为 ( D, )的运行时方差。
Returns 元组:
- out: 输出( N, D )。
- cache: 用于反向传播的缓存。
"""
mode = bn_param[ 'mode' ]
eps = bn_param.get( 'eps', 1e-5 )
momentum = bn_param.get( 'momentum', 0.9 )
N, D = x.shape
running_mean = bn_param.get( 'running_mean', np.zeros( D, dtype = x.dtype ) )
running_var = bn_param.get( 'running_var', np.zeros( D, dtype = x.dtype ) )
out, cache = None, None
if mode == 'train':
  ##########################################################################
  #                  任务：实现训练阶段 BN 的前向传播。                  #
  #        首先，你需要计算输入数据的均值和方差；                        #
  #        然后，使用均值和方差将数据进行归一化处理；                    #
  #        之后，使用 gamma 和 beta 参数将数据进行缩放和平移；           #
  #        最后，将该批数据均值和方差添加到累积均值和方差中；            #
  #        注意：将反向传播时所需的所有中间值保存在 cache 中。           #
  ##########################################################################
```

```
    ##############################################################
    #                          结束编码                            #
    ##############################################################
elif mode == 'test':
    ##############################################################
    #            任务：实现测试阶段 BN 的前向传播。                  #
    #        首先，使用运行时均值与方差归一化数据，                  #
    #        然后，使用 gamma 和 beta 参数缩放，平移数据。           #
    ##############################################################




    ##############################################################
    #                          结束编码                            #
    ##############################################################
else:
    raise ValueError( '无法识别的 BN 模式： "%s"' % mode )
# 更新运行时均值，方差。
bn_param[ 'running_mean' ] = running_mean
bn_param[ 'running_var' ] = running_var
return out, cache
```

完成上述代码后，我们使用下列代码检验已实现的 BN 前向传播。BN 处理后均值应该接近于 0，标准差应该接近于 1。再经过 gamma 和 beta 的缩放与平移后，均值应该接近于 beta，标准差应该接近于 gamma。

训练阶段，BN 前向传播的均值与标准差检验代码块：

```
# 检验 BN 训练阶段前向传播的均值和标准差。
N, D1, D2, D3 = 200, 50, 60, 3
X = np.random.randn( N, D1 )
W1 = np.random.randn( D1, D2 )
W2 = np.random.randn( D2, D3 )
a = np.maximum( 0, X.dot( W1 ) ).dot( W2 )
print 'normalization 之前:'
print '    means: ', a.mean( axis = 0 )
print '    stds: ', a.std( axis = 0 )
# 均值应该接近零，标准差接近 1。
print 'batch normalization 之后 （gamma = 1, beta = 0 )'
```

```
a_norm, _ = batchnorm_forward( a, np.ones( D3 ), np.zeros( D3 ), { 'mode': 'train' } )
print '   mean: ', a_norm.mean( axis = 0 )
print '   std: ', a_norm.std( axis = 0 )
# 均值应该接近 beta，标准差接近 gamma。
gamma = np.asarray( [ 1.0, 2.0, 3.0 ] )
beta = np.asarray( [ 11.0, 12.0, 13.0 ] )
a_norm, _ = batchnorm_forward( a, gamma, beta, { 'mode': 'train' } )
print ' batch normalization 之后 （ 随机 gamma, beta )'
print '   means: ', a_norm.mean( axis = 0 )
print '   stds: ', a_norm.std( axis = 0 )
```

正确编码后的执行结果：

```
normalization 之前:
  means:  [ 10.04031787   11.64030966 -20.10740986 ]
  stds:  [ 27.88819957   39.07733242   35.78698193 ]
batch normalization 之后 （ gamma = 1, beta = 0 )
  mean:  [ -6.88338275e-17   -1.09426357e-16     1.08142661e-16 ]
  std:  [ 0.99999999   1.            1.          ]
 batch normalization 之后 （ 随机 gamma, beta )
  means:  [ 11.   12.   13. ]
  stds:  [ 0.99999999   1.99999999   2.99999999 ]
```

接下来，我们验证 BN 算法测试模式时的代码。需要注意，由于我们使用运行时平均，因此需要训练一段时间后，运行结果才会稳定。

测试阶段，BN 算法前向传播检验代码块：

```
# 检验测试阶段前向传播。
# 注意：需要训练一段时间后，运行时均值才会稳定。
N, D1, D2, D3 = 200, 50, 60, 3
W1 = np.random.randn( D1, D2 )
W2 = np.random.randn( D2, D3 )
bn_param = { 'mode': 'train' }
gamma = np.ones( D3 )
beta = np.zeros( D3 )
for t in xrange( 50 ) :
  X = np.random.randn( N, D1 )
  a = np.maximum( 0, X.dot( W1 ) ).dot( W2 )
  batchnorm_forward( a, gamma, beta, bn_param )
bn_param[ 'mode' ] = 'test'
X = np.random.randn( N, D1 )
```

```
a = np.maximum( 0, X.dot( W1 ) ).dot( W2 )
a_norm, _ = batchnorm_forward( a, gamma, beta, bn_param )
# 均值应该接近 0，标准差接近 1。由于使用运行时均值，可能会带有一点噪声。
print 'batch normalization 之后 （ 测试阶段 ):'
print '    means: ', a_norm.mean( axis = 0 )
print '    stds: ', a_norm.std( axis = 0 )
```

测试模式时的 BN 算法检验结果：

```
batch normalization 之后 （ 测试阶段 ):
  means:   [ 0.22538848   -0.04480854   -0.0726637 ]
  stds:  [ 0.96726806      1.06688483     0.93590139 ]
```

5.8.6 BN 反向传播

接下来完成 BN 算法的反向传播，可能需要用到前向传播时中间结果的所有缓存。静下心来慢慢理顺该部分内容，如果完成了记得给自己一顿奖赏。

batchnorm_backward 函数代码块：

```
def batchnorm_backward(dout, cache):
  """
  BN 反向传播。
  Inputs:
  - dout: 上层梯度 （ N, D )。
  - cache: 前向传播时的缓存。
  Returns 元组:
  - dx: 数据梯度( N, D )。
  - dgamma: gamma 梯度( D, )。
  - dbeta: beta 梯度( D, )。
  """
  dx, dgamma, dbeta = None, None, None
  #############################################################################
  #                       任务：实现 BN 反向传播。                              #
  #                 将结果分别保存在 dx，dgamma，dbeta 中。                     #
  #############################################################################



  #############################################################################
  #                                结束编码                                   #
  #############################################################################
  return dx, dgamma, dbeta
```

完成上述代码后，使用下列代码检验 BN 反向传播梯度。

检验 BN 反向传播梯度代码块：

```
# 检验 BN 反向传播梯度。
N, D = 4, 5
x = 5 * np.random.randn( N, D ) + 12
gamma = np.random.randn( D )
beta = np.random.randn( D )
dout = np.random.randn( N, D )
bn_param = { 'mode': 'train' }
fx = lambda x: batchnorm_forward( x, gamma, beta, bn_param )[ 0 ]
fg = lambda a: batchnorm_forward( x, gamma, beta, bn_param )[ 0 ]
fb = lambda b: batchnorm_forward( x, gamma, beta, bn_param )[ 0 ]
dx_num = eval_numerical_gradient_array( fx, x, dout )
da_num = eval_numerical_gradient_array( fg, gamma, dout )
db_num = eval_numerical_gradient_array( fb, beta, dout )
_, cache = batchnorm_forward( x, gamma, beta, bn_param )
dx, dgamma, dbeta = batchnorm_backward( dout, cache )
print 'dx  误差: ', rel_error( dx_num, dx )
print 'dgamma 误差: ', rel_error( da_num, dgamma )
print 'dbeta  误差: ', rel_error( db_num, dbeta )
```

正确编码后的检验结果：

```
dx  误差:   2.70249386048e-09
dgamma 误差:   1.84440012476e-11
dbeta  误差:   8.31301927287e-12
```

- BN 反向传播（公式）

之前的内容中，我们将 BN 算法的反向传播过程拆解为很多个中间值，这样做逻辑很清晰，但大大地降低了执行效率。接下来，我们直接使用推导公式计算 dx，你可以自己手动推导下 BN 的反向传播，然后比较下两者的执行效率。这部分代码已经编写好了，阅读下面代码直接使用即可。

batchnorm_backward_alt 函数代码块：

```
def batchnorm_backward_alt( dout, cache ) :
  """
  可选的 BN 反向传播。
  """
  dx, dgamma, dbeta = None, None, None
  mu, xmu, carre, var, sqrtvar, invvar, va2, va3, gamma, beta, x, bn_param = cache
```

```
eps = bn_param.get( 'eps', 1e-5 )
N, D = dout.shape
dbeta = np.sum( dout, axis = 0 )
dgamma = np.sum( ( x – mu ) * ( var + eps )**( -1. / 2. ) * dout, axis = 0 )
dx = ( 1./ N ) * gamma * ( var + eps )**( -1./ 2. ) * ( N * dout - np.sum(
        dout, axis = 0 ) - ( x – mu ) * ( var + eps )**( -1.0 ) * np.sum( dout * ( x -mu ), axis = 0 ) )
return dx, dgamma, dbeta
```

公式法 BN 反向传播测试代码块：

```
N, D = 100, 500
x = 5 * np.random.randn( N, D ) + 12
gamma = np.random.randn( D )
beta = np.random.randn( D )
dout = np.random.randn( N, D )
bn_param = { 'mode': 'train' }
out, cache = batchnorm_forward( x, gamma, beta, bn_param )
t1 = time.time( )
dx1, dgamma1, dbeta1 = batchnorm_backward( dout, cache )
t2 = time.time( )
dx2, dgamma2, dbeta2 = batchnorm_backward_alt( dout, cache )
t3 = time.time( )
print 'dx 误差: ', rel_error( dx1, dx2 )
print 'dgamma 误差: ', rel_error( dgamma1, dgamma2 )
print 'dbeta 误差: ', rel_error( dbeta1, dbeta2 )
print '加速: %.2fx' % ( ( t2 - t1 ) / ( t3 - t2 ) )
```

正确编码后的执行结果：

```
dx 误差:   2.50238754792e-12
dgamma 误差:   4.06951863454e-14
dbeta 误差:   0.0
加速: 3.00x
```

5.8.7 使用 BN 的全连接网络

接下来，打开“DLAction/classifiers/chapter5/fc_net.py”文件，实现 BN 算法的全连接网络。现在我们的网络将有以下 4 种选择。

- 深层全连接;
- 全连接+dropout;
- 全连接+BN;

- 全连接+BN+dropout。

可以先实现 affine_BN_relu 函数和 affine_BN_relu_drop 函数，这样有利于降低编码复杂度。

网络初始化代码块：

```
def __init__( self, input_dim = 3 * 32 * 32, hidden_dims=[ 100 ], num_classes = 10,
              dropout = 0, use_batchnorm = False, reg = 0.0,
              weight_scale = 1e-2, seed = None ) :
    """
    初始化全连接网络。
    Inputs:
    - input_dim: 输入维度。
    - hidden_dims: 隐藏层各层维度向量，如[ 100,100 ]。
    - num_classes: 分类个数。
    - dropout: 如果 dropout = 0，表示不使用 dropout。
    - use_batchnorm：布尔型，表示是否使用 BN。
    - reg:正则化衰减因子。
    - weight_scale:权重初始化范围，标准差。
    - seed: 使用 seed 产生相同的随机数。
    """
    self.use_batchnorm = use_batchnorm
    self.use_dropout = dropout > 0
    self.reg = reg
    self.num_layers = 1 + len( hidden_dims )
    self.params = { }
    ############################################################################
    #                  任务：初始化网络参数。                                    #
    #          权重参数初始化和前面章节类似，                                    #
    #          针对每一层神经元都要初始化对应的 gamma 和 beta。                  #
    #          如：第一层使用 gamma1，beta1，第二层 gamma2，beta2，              #
    #          gamma 初始化为 1，beta 初始化为 0。                               #
    ############################################################################



    ############################################################################
    #                             结束编码                                       #
    ############################################################################
```

```
    self.dropout_param = { }
    if self.use_dropout:
      self.dropout_param = { 'mode': 'train', 'p': dropout }
      if seed is not None:
        self.dropout_param[ 'seed' ] = seed
    self.bn_params = [ ]
    if self.use_batchnorm:
      self.bn_params = [ { 'mode': 'train' } for i in xrange( self.num_layers – 1 ) ]
```

损失函数代码块：

```
  def loss(self, X, y=None):
    # 设置执行模式。
    mode = 'test' if y is None else 'train'
    if self.dropout_param is not None:
      self.dropout_param[ 'mode' ] = mode
    if self.use_batchnorm:
      for bn_param in self.bn_params:
        bn_param[ mode ] = mode
    scores = None
    ############################################################################
    #                任务：执行全连接网络的前馈过程。                          #
    #             计算数据的分类得分，将结果保存在 scores 中。                 #
    #  当使用 dropout 时，需要使用 self.dropout_param 进行 dropout 前馈。      #
    #  当使用 BN 时，self.bn_params[0]传到第一层，                             #
    #                  self.bn_params[1]传到第二层。                           #
    ############################################################################



    ############################################################################
    #                               结束编码                                   #
    ############################################################################
    if mode == 'test':
      return scores
    loss, grads = 0.0, { }
    ############################################################################
    #           任务：实现全连接网络的反向传播。                               #
```

```
#          将损失值存储在 loss 中，梯度值存储在 grads 字典中。          #
#          当使用 dropout 时，需要求解 dropout 梯度。                  #
#          当使用 BN 时，需要求解 BN 梯度。                            #
##########################################################################




##########################################################################
#                              结束编码                                  #
##########################################################################
return loss, grads
```

BN 全连接网络验证代码块：

```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn( N, D )
y = np.random.randint( C, size = ( N, ) )
for reg in [ 0, 3.14 ]:
  print '检验  reg = ', reg
  model = FullyConnectedNet( hidden_dims = [ H1, H2 ], input_dim = D, num_classes = C,
                             reg = reg, weight_scale = 5e-2, use_batchnorm = True )
  loss, grads = model.loss( X, y )
  print '初始化  loss: ', loss
  for name in sorted( grads ) :
    f = lambda _: model.loss( X, y )[ 0 ]
grad_num = eval_numerical_gradient( f, model.params[ name ],
verbose = False, h = 1e-5 )
    print '%s  相对误差: %.2e' % ( name, rel_error(grad_num, grads[ name ] ) )
```

无权重衰减结果：	有权重衰减验证结果：
检验 reg = 0	检验 reg = 3.14
初始化 loss: 2.35632164622	初始化 loss: 7.11059525798
W1 相对误差: 2.53e-04	W1 相对误差: 1.26e-06
W2 相对误差: 1.53e-05	W2 相对误差: 5.23e-07
W3 相对误差: 3.80e-10	W3 相对误差: 7.31e-09
b1 相对误差: 2.22e-03	b1 相对误差: 8.88e-07
b2 相对误差: 6.27e-07	b2 相对误差: 5.77e-07
b3 相对误差: 9.30e-11	b3 相对误差: 4.00e-10
beta1 相对误差: 8.16e-09	beta1 相对误差: 8.38e-09

beta2 相对误差: 3.20e-09	beta2 相对误差: 4.07e-08
gamma1 相对误差: 1.11e-08	gamma1 相对误差: 8.10e-09
gamma2 相对误差: 2.67e-09	gamma2 相对误差: 3.72e-08

接下来，我们测试加入 BN 与不加入 BN 对于网络的影响，运行下列代码。图 5-14、图 5-15 与图 5-16 分别为加入 BN 与不加入 BN 两种算法的损失函数变化、训练精度变化和验证精度变化对比示意图。

使用 BN 训练深层神经网络代码块：

```
# 使用 BN 训练深层神经网络。
hidden = [ 100, 100, 100, 100, 100 ]
num_train = 1000
small_data = {
  'X_train': data[ 'X_train' ][ : num_train ],
  'y_train': data[ 'y_train' ][ : num_train ],
  'X_val': data[ 'X_val' ],
  'y_val': data[ 'y_val' ],
}
weight_scale = 2e-2
bn_model = FullyConnectedNet( hidden_dims = hidden,
weight_scale = weight_scale, use_batchnorm = True )
model = FullyConnectedNet( hidden_dims = hidden,
weight_scale = weight_scale, use_batchnorm = False )
bn_trainer = Trainer( bn_model, small_data,
                num_epochs = 10, batch_size = 50, update_rule = 'adam',
                updater_config = { 'learning_rate': 1e-3, },
                verbose = True, print_every = 200 )
bn_trainer.train( )
trainer = Trainer( model, small_data,
                num_epochs = 10, batch_size = 50, update_rule = 'adam',
                updater_config = { 'learning_rate': 1e-3, },
                verbose = True, print_every = 200 )
trainer.train( )
```

可视化结果代码块：

```
plt.subplots_adjust( left = 0.08, right = 0.95, wspace = 0.25, hspace = 0.3 )
plt.subplot( 3, 1, 1 )
plt.title( 'Training loss', fontsize = 18 )
plt.xlabel( 'Iteration', fontsize = 18 )
plt.ylabel( 'Loss', fontsize = 18 )
```

```
plt.subplot( 3, 1, 2 )
plt.title( 'Training accuracy', fontsize = 18 )
plt.xlabel( 'Epoch', fontsize = 18 )
plt.ylabel( 'Accuracy', fontsize = 18 )
plt.subplot( 3, 1, 3 )
plt.title( 'Validation accuracy', fontsize = 18 )
plt.xlabel( 'Epoch', fontsize = 18 )
plt.ylabel( 'Accuracy', fontsize = 18 )
plt.subplot( 3, 1, 1 )
plt.plot( trainer.loss_history, '*', label = 'baseline' )
plt.plot( bn_trainer.loss_history, 'D', label = 'batchnorm' )
plt.subplot( 3, 1, 2 )
plt.plot( trainer.train_acc_history, '-*', label = 'baseline' )
plt.plot( bn_trainer.train_acc_history, '-D', label = 'batchnorm' )
plt.subplot( 3, 1, 3 )
plt.plot( trainer.val_acc_history, '-*', label = 'baseline' )
plt.plot( bn_trainer.val_acc_history, '-D', label = 'batchnorm' )
for i in [ 1, 2, 3 ]:
  plt.subplot( 3, 1, i )
  plt.legend( loc = 'upper center', ncol = 4 )
plt.gcf( ).set_size_inches( 15, 15 )
plt.show( )
```



图 5-14　BN 神经网络与标准神经网络损失函数变化比较示意图

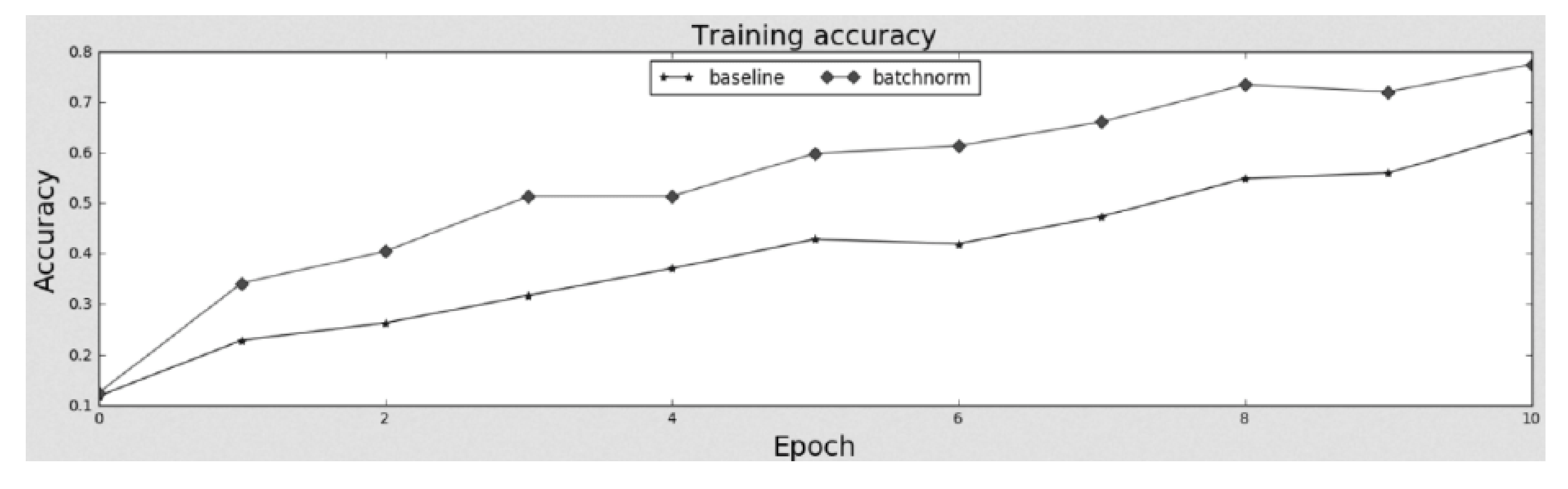


图 5-15　BN 神经网络与标准神经网络训练精度变化比较示意图

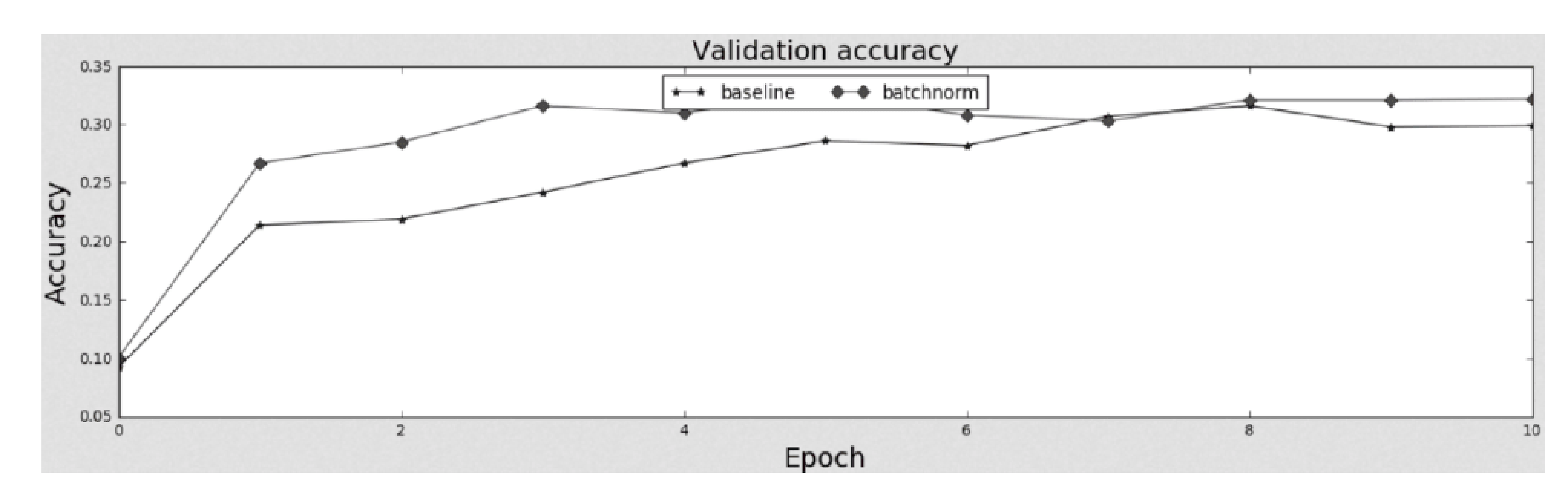


图 5-16　BN 神经网络与标准神经网络验证精度变化比较示意图

由图 5-14 至图 5-16 可看出，BN 算法加快了网络的训练效率，并且拥有更佳的验证精度。

5.8.8　BN 算法与权重标准差比较

在 5.6 参数初始化章节中，我们提到过权重参数的标准差，显著地影响着网络的训练。在本小节中，我们将比较不同权重规模下 BN 算法如何提升网络性能，并且 BN 算法还大大降低了不同权重初始化所带来的影响。运行下列代码块，图 5-17、图 5-18 与图 5-19 分别为 BN 算法在不同权重初始化情况下的最佳验证精度、训练精度和损失值可视化示意图。

BN 与权重初始化比较代码块：

```
hidden = [ 50, 50, 50, 50, 50, 50, 50 ]
num_train = 1000
small_data = {
  'X_train': data[ 'X_train' ][ : num_train ],
  'y_train': data[ 'y_train' ][ : num_train ],
  'X_val': data[ 'X_val' ],
  'y_val': data[ 'y_val' ],
}
bn_trainers = { }
trainers = { }
weight_scales = np.logspace( -4, 0, num = 20 )
t1 = time.time( )
for i, weight_scale in enumerate( weight_scales ):
  print 'Running weight scale %d / %d' % ( i + 1, len( weight_scales ) )
  bn_model = FullyConnectedNet( hidden_dims = hidden,
weight_scale = weight_scale, use_batchnorm = True )
  model = FullyConnectedNet(hidden_dims = hidden,
weight_scale = weight_scale, use_batchnorm = False )
  bn_trainer = Trainer( bn_model, small_data, num_epochs = 10, batch_size = 50,
                    update_rule = 'adam',   updater_config = { 'learning_rate': 3e-3, },
                    verbose = False, print_every = 200 )
```

```
    bn_trainer.train( )
    bn_trainers[ weight_scale ] = bn_trainer
    trainer = Trainer( model, small_data,   num_epochs = 10, batch_size = 50,
                       update_rule = 'adam', updater_config = { 'learning_rate': 3e-3, },
                       verbose = False, print_every = 200 )
    trainer.train( )
    trainers[ weight_scale ] = trainer
t2 = time.time( )
print 'time: %.2f' % ( t2 - t1 )
```

可视化训练结果：

```
best_train_accs, bn_best_train_accs = [ ], [ ]
best_val_accs, bn_best_val_accs = [ ], [ ]
final_train_loss, bn_final_train_loss = [ ], [ ]
for ws in weight_scales:
    best_train_accs.append( max( solvers[ ws ].train_acc_history ) )
    bn_best_train_accs.append( max( bn_solvers[ ws ].train_acc_history ) )
    best_val_accs.append( max( solvers[ ws ].val_acc_history ) )
    bn_best_val_accs.append( max( bn_solvers[ ws ].val_acc_history ) )
    final_train_loss.append( np.mean( solvers[ ws ].loss_history[ -100 : ] ) )
    bn_final_train_loss.append( np.mean( bn_solvers[ ws ].loss_history[ -100 : ] ) )
plt.subplots_adjust( left = 0.08, right = 0.95, wspace = 0.25, hspace = 0.3 )
plt.subplot( 3, 1, 1 )
plt.title( 'Best val accuracy vs weight initialization scale', fontsize = 18 )
plt.xlabel( 'Weight initialization scale', fontsize = 18 )
plt.ylabel( 'Best val accuracy', fontsize = 18 )
plt.semilogx( weight_scales, best_val_accs, '-D', label = 'baseline' )
plt.semilogx( weight_scales, bn_best_val_accs, '-*', label = 'batchnorm' )
plt.legend( ncol = 2, loc = 'lower right' )
plt.subplot( 3, 1, 2 )
plt.title( 'Best train accuracy vs weight initialization scale', fontsize = 18 )
plt.xlabel ('Weight initialization scale', fontsize = 18 )
plt.ylabel( 'Best training accuracy',fontsize = 18 )
plt.semilogx( weight_scales, best_train_accs, '-D', label = 'baseline' )
plt.semilogx( weight_scales, bn_best_train_accs, '-*', label = 'batchnorm' )
plt.legend( )
plt.subplot(3, 1, 3)
plt.title( 'Final training loss vs weight initialization scale', fontsize = 18 )
plt.xlabel( 'Weight initialization scale', fontsize = 18 )
```

```
plt.ylabel( 'Final training loss', fontsize = 18 )
plt.semilogx( weight_scales, final_train_loss, '-D', label = 'baseline' )
plt.semilogx( weight_scales, bn_final_train_loss, '-*', label = 'batchnorm' )plt.legend( )
plt.gcf( ).set_size_inches( 10, 15 )
plt.show( )
```

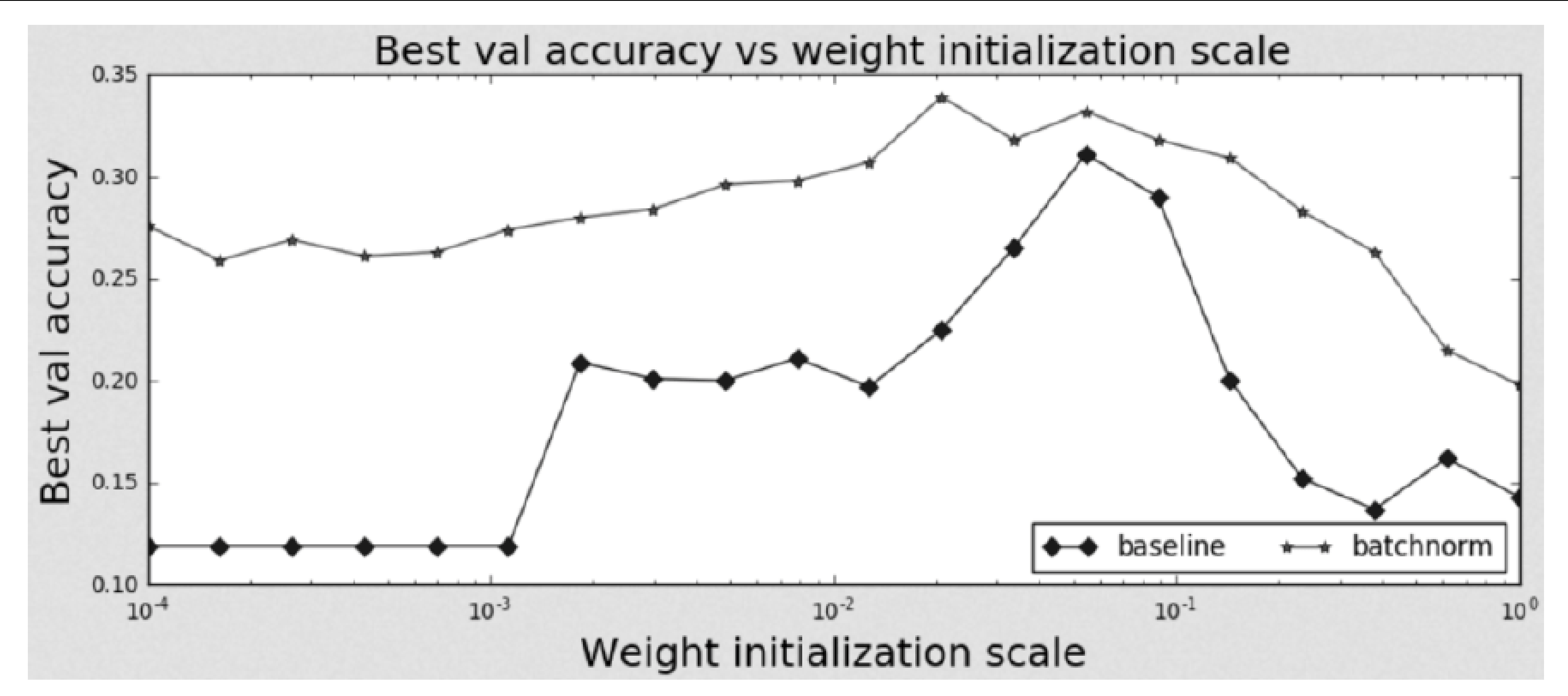


图 5-17 BN 算法在不同权重初始化情况下的最佳验证精度

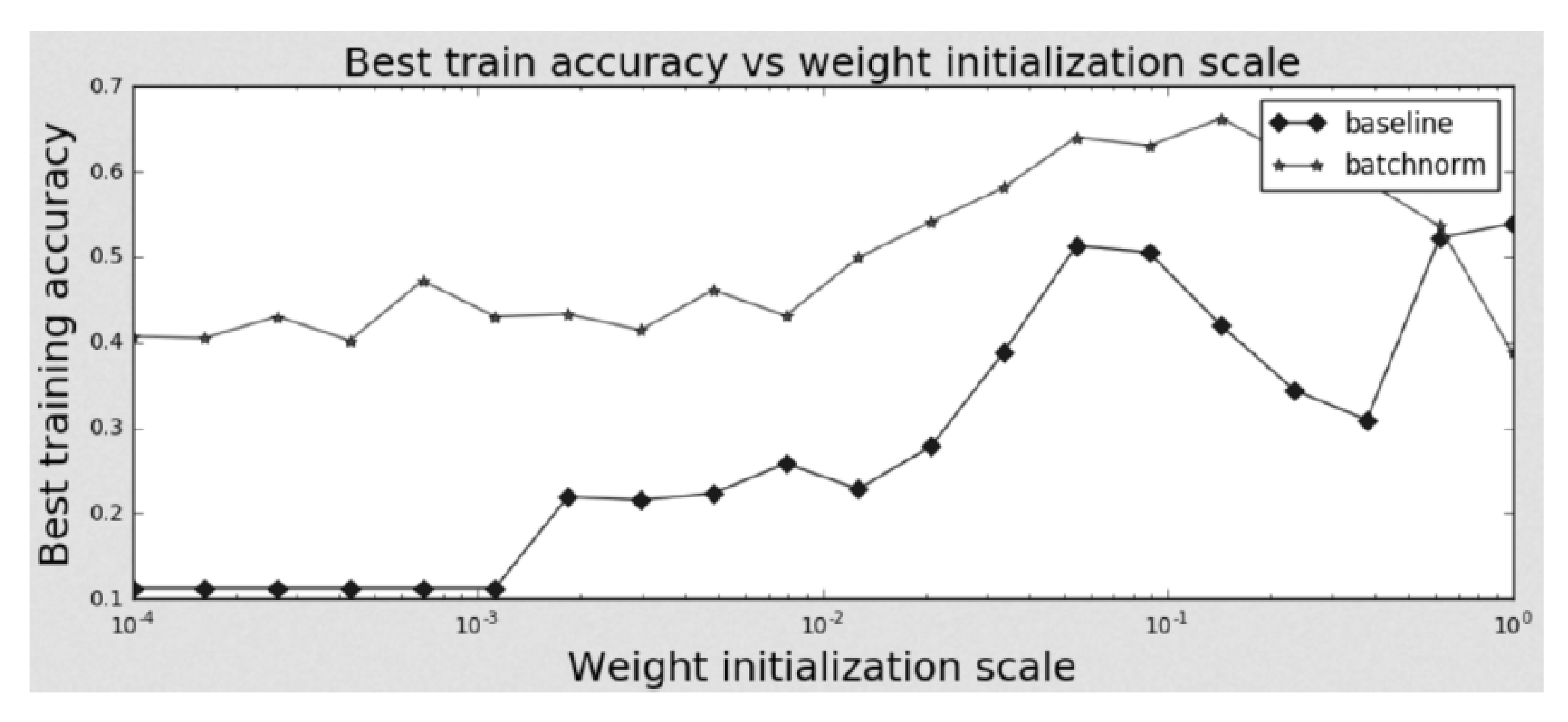


图 5-18 BN 算法在不同权重初始化情况下的最佳训练精度

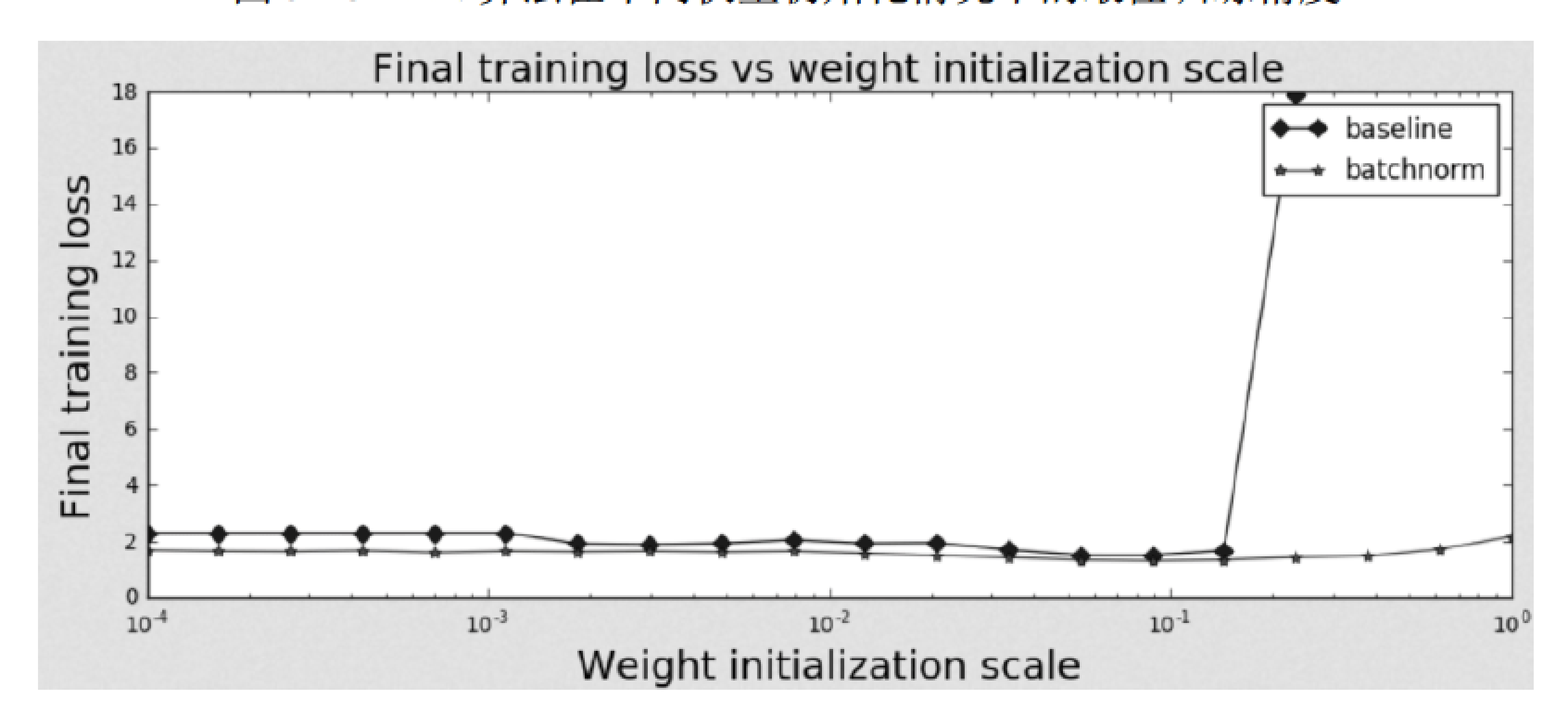


图 5-19 BN 算法在不同权重初始化情况下的损失值

由图 5-17 至图 5-19 可以看出，BN 算法显著地降低了权重初始化对深层网络的影响。并且在绝大多数情况下，加入 BN 操作都提升了网络性能。

5.9　参考代码

动量更新规则代码模块：

```
def sgd_momentum( w, dw, config = None ) :
    v =config[ 'momentum' ] * config[ 'velocity' ] - config[ 'learning_rate' ] * dw
    next_w = w + v
```

RMSProp 更新规则代码块：

```
def rmsprop( w, dw, config = None):
    config[ 'cache' ] = config[ 'decay_rate' ] * config[ 'cache' ] + ( 1 - config[ 'decay_rate' ] ) * dw **2
    next_w = w - config[ 'learning_rate' ] * dw / ( np.sqrt( config[ 'cache' ] + config[ 'epsilon' ] ) )
    return next_w, config
```

Adam 更新函数代码块：

```
def adam( w, dw, config = None ) :
    config[ 't' ] + = 1
    beta1 = config[ 'beta1' ]
    beta2 = config[ 'beta2' ]
    epsilon = config[ 'epsilon' ]
    learning_rate = config[ 'learning_rate' ]
    config[ 'm' ] = beta1 * config[ 'm' ] + ( 1 - beta1 ) * dw
    config[ 'v' ] = beta2 * config[ 'v' ] + ( 1 - beta2 ) * dw **2
    mb = config[ 'm' ] / ( 1 - beta1 **config[ 't' ] )
    vb = config[ 'v' ] / ( 1 - beta2 **config[ 't' ] )
    next_w = w - learning_rate * mb / ( np.sqrt( vb ) + epsilon )
    return next_w, config
```

batchnorm_forward 函数代码块：

```
def batchnorm_forward ( x, gamma, beta, bn_param ) :
    if mode == 'train':
        mu = 1 / float( N ) * np.sum( x, axis = 0 )
        xmu = x - mu
        carre = xmu **2
        var = 1 / float( N ) * np.sum( carre, axis = 0 )
        sqrtvar = np.sqrt( var + eps )
        invvar = 1. / sqrtvar
        va2 = xmu * invvar
        va3 = gamma * va2
        out = va3 + beta
```

```
    running_mean = momentum * running_mean + (1.0 – momentum ) * mu
    running_var = momentum * running_var + (1.0 – momentum ) * var
    cache = ( mu, xmu, carre, var, sqrtvar, invvar,va2, va3, gamma, beta, x, bn_param )
  elif mode == 'test':
    mu = running_mean
    var = running_var
    xhat = (x - mu) / np.sqrt( var + eps )
    out = gamma * xhat + beta
    cache = ( mu, var, gamma, beta, bn_param )
  else:
    raise ValueError( '无法识别的 BN 模式：  "%s"' % mode )
  bn_param[ 'running_mean' ] = running_mean
  bn_param[ 'running_var' ] = running_var
  return out, cache
```

batchnorm_backward 函数代码块：

```
def batchnorm_backward( dout, cache ) :
  dx, dgamma, dbeta = None, None, None
  mu, xmu, carre, var, sqrtvar, invvar, va2, va3, gamma, beta, x, bn_param = cache
  eps = bn_param.get( 'eps', 1e-5 )
  N, D = dout.shape
  # 第 9 步反向传播
  dva3 = dout
  dbeta = np.sum( dout, axis = 0 )
  # 第 8 步反向传播
  dva2 = gamma * dva3
  dgamma = np.sum( va2 * dva3, axis = 0 )
  # 第 7 步反向传播
  dxmu = invvar * dva2
  dinvvar = np.sum( xmu * dva2, axis = 0 )
  # 第 6 步反向传播
  dsqrtvar = -1. / ( sqrtvar **2 ) * dinvvar
  # 第 5 步反向传播
  dvar = 0.5 * ( var + eps )**( -0.5 ) * dsqrtvar
  # 第 4 步反向传播
  dcarre = 1 / float( N ) * np.ones( ( carre.shape ) ) * dvar
  # 第 3 步反向传播
  dxmu + = 2 * xmu * dcarre
  # 第 2 步反向传播
```

```
    dx = dxmu
    dmu = - np.sum( dxmu, axis = 0 )
    # 第 1 步反向传播
    dx += 1 / float( N ) * np.ones( ( dxmu.shape ) ) * dmu
    return dx, dgamma, dbeta
```

网络初始化代码模块：

```
def __init__( self, input_dim = 3 * 32 * 32, hidden_dims = [ 100 ], num_classes = 10, dropout = 0,
              use_batchnorm = False, reg = 0.0, weight_scale = 1e-2, seed = None ) :
    layers_dims = [ input_dim ] + hidden_dims + [ num_classes ]
    for i in xrange( self.num_layers ) :
        self.params[ 'W' + str( i + 1 ) ] = weight_scale * np.random.randn(
            layers_dims[ i ], layers_dims[ i + 1 ] )
        self.params[ 'b' + str( i + 1 ) ] = np.zeros( ( 1, layers_dims[ i + 1 ] ) )
        if self.use_batchnorm and i < len( hidden_dims ) :
            self.params[ 'gamma' + str( i + 1 ) ] = np.ones( ( 1, layers_dims[ i + 1 ] ) )
            self.params[ 'beta' + str( i + 1 ) ] = np.zeros( ( 1, layers_dims[ i + 1 ] ) )
```

损失函数代码块：

```
  def loss( self, X, y = None ) :
    scores = None
    outs,cache = { },{ }
    outs[ 0 ] = X
    num_h = self.num_layers - 1
    for i in xrange( num_h ) :
        if self.use_dropout:
            outs[ i + 1 ], cache[ i + 1 ] = affine_relu_dropout_forward(
                outs[ i ], self.params[ 'W' + str( i + 1 ) ],
                self.params[ 'b' + str( i + 1 ) ], self.dropout_param )
        elif self.use_batchnorm :
            gamma = self.params[ 'gamma' + str( i + 1 ) ]
            beta = self.params[ 'beta' + str( i + 1 ) ]
            outs[ i + 1 ], cache[ i + 1 ] = affine_bn_relu_forward(
                                            outs[ i ], self.params[ 'W' + str( i + 1 ) ],
                                            self.params[ 'b' + str( i + 1 ) ],
                                            gamma, beta, self.bn_params[ i ] )
        else :
            outs[ i + 1 ], cache[ i + 1 ] = affine_relu_forward(
                                             outs[ i ] ,
```

```
                                    self.params[ 'W' + str( i + 1 ) ],
                                    self.params[ 'b' + str( i + 1 ) ] )
scores, cache[ num_h + 1 ] = affine_forward(
                                    outs[ num_h ],
                                    self.params[ 'W' + str( num_h + 1 ) ],
                                    self.params[ 'b' +str( num_h + 1 ) ] )
if mode == 'test':
  return scores
loss, grads = 0.0, { }
dout = { }
loss,dy = softmax_loss( scores, y )
h = self.num_layers -1
for i in xrange( self.num_layers ):
    loss + = 0.5 * self.reg * ( np.sum(
            self.params[ 'W' + str( i + 1 ) ] * self.params[ 'W' + str( i + 1 ) ] ) )
dout[ h ], grads[ 'W' + str( h + 1 ) ], grads[ 'b' + str( h + 1 ) ] = affine_backward( dy, cache[ h + 1 ] )
grads[ 'W' + str( h + 1 ) ] + = self.reg * self.params[ 'W' + str( h + 1 ) ]
for i in xrange( h ) :
    if self.use_dropout:
        dx,dw,db = affine_relu_dropout_backward( dout[ h – i ], cache[ h – i ] )
        dout[ h - 1 – i ] = dx
        grads[ 'W' + str( h – i ) ] = dw
        grads[ 'b' + str( h - i ) ] = db
    elif self.use_batchnorm:
        dx, dw, db, dgamma, dbeta = affine_bn_relu_backward( dout[ h – i ], cache[ h – i ] )
        dout[ h – 1 – i ] = dx
        grads[ 'W' + str( h – i ) ] = dw
        grads[ 'b' + str( h – i ) ] = db
        grads[ 'gamma' + str( h – i ) ] = dgamma
        grads[ 'beta' + str( h – i ) ] = dbeta
    else :
        dx, dw, db = affine_relu_backward( dout[ h – i ], cache[ h – i ] )
        dout[ h - 1 – i ] = dx
        grads[ 'W' + str( h – i ) ] = dw
        grads[ 'b' + str( h – i ) ] = db
    grads[ 'W' + str( h – i ) ] + = self.reg * self.params[ 'W' + str ( h – i ) ]
return loss, grads
```

5.10　参考文献

[1] Sontag, E. D., & Sussmann, H. J. (1997). Backpropagation Can Give Rise To Spurious Local Minima Even For Networks Without Hidden Layers. Complex Systems, 3(1), 91-106.

[2] Goodfellow, I. J., Vinyals, O., & Saxe, A. M. (2015). Qualitatively characterizing neural network optimization problems. Computer Science.

[3] Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., & Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. Mathematics, 111(6 Pt 1), 2475-2485.

[4] Pascanu, R., Mikolov, T., & Bengio, Y. (2013). On the difficulty of training recurrent neural networks. Paper presented at the International Conference on International Conference on Machine Learning.

[5] Hochreiter, S. (2011). The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. 6(2), 107-116.

[6] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. Paper presented at the International Conference on Neural Information Processing Systems.

[7] Pascanu, R., Mikolov, T., & Bengio, Y. (2012). Understanding the exploding gradient problem. Arxiv Preprint Arxiv.

[8] Blum, A., & Rivest, R. L. (1988). Training a 3-node neural network is NP-complete. Paper presented at the The Workshop on Computational Learning Theory.

[9] Bottou, L. (1998). Online Algorithms and Stochastic Approximations.

[10] Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. Ussr Computational Mathematics & Mathematical Physics, 4(5), 1-17.

[11] Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12(7), 2121-2159.

[12] Hinton, G. E. (2012). Tutorial on deep learning. IPAM Graduate Summer School: Deep Learning, Feature Learning. 261

[13] Kingma, D., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. Computer Science.

[14] Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks. Paper presented at the International Conference on Artificial Intelligence and Statistics.

[15] Martens, J. (2010). Deep learning via Hessian-free optimization. Paper presented at the International Conference on Machine Learning.

[16] Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2006). Greedy layer-wise training of deep networks. Paper presented at the International Conference on Neural Information Processing Systems.

[17] Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Computer Science.

第 6 章

卷积神经网络

卷积神经网络（Convolutional Neural Network，CNN）[1]是在实际应用中最为成功的一种神经网络，其专门用于处理格状结构数据，比如图片数据就可以看成是由像素组成的二维格状数据。之所以称为“卷积”是因为其数据处理方式类似于数学中的卷积操作，该网络也是人工智能领域受生物启发最成功的模型之一，该模型几乎垄断了机器视觉方面的研究，本章我们将详细地讲解该模型。

在 6.1 节中，我们将从数学的角度介绍什么是**卷积操作**，并介绍数学中的“卷积”与机器学习中“卷积”的异同。

在 6.2 节中，我们将从哺乳动物视神经开始介绍**卷积的意义**，然后再从生物学迁移到机器学习，介绍卷积在机器学习中的具体优势：**稀疏连接**与**参数共享**。

在 6.3 节中，我们将介绍**池化**（Pooling）操作是如何降低模型复杂度的同时还可以获取数据的空间不变性，平移不变性等特征。

在 6.4 节中，我们将介绍如何在卷积网络中加入**跨步卷积**、**零填充**、**局部连接和平铺连接**等方法来设计特定的卷积网络。

在 6.5 节中，我们将完成卷积神经网络的编码练习，我们将逐步学习编码卷积、池化等操作，然后再将其组合成完整的卷积神经网络。

6.1　卷积操作

传说杜小飞是一个高级间谍，在执行任务时发现了敌人的惊天阴谋但却无法脱身。他只能用一种神奇的墨水，将情报写入一幅**很大**的油画中，然后托人将其带到长官手中。长官拥有神奇的**小**手电筒，可以将油画中的秘密信息照出来。那么请问，长官如何才能将油画中的内容照出来呢？

答案是从油画的左上角开始，从左向右，从上到下，一直扫描到油画的右下角。希望你看到以上内容时没有口吐鲜血，然后把本书撕了。因为上述内容确实就是卷积网络所做的事情。那把神奇的小手电筒就是**卷积核**（Kernel），而扫描出的内容，也就是卷积的输出结果，也称为**特征映射或特征图**（Feature Map）。那卷积又是什么呢？简单来说，卷积其实就是对**数据加权求和**。

如果小燕儿同学犯了一个错误，杨老师就会用皮尺打小燕儿的手，那么小燕儿的手就会立刻浮肿起来。但过段时间，手也会渐渐地痊愈。现在我们就做一个残忍的假设：假设小燕儿以一个时间频率不断地犯错，那杨老师也就会以一个时间频率不断地打小燕儿的小手。旧的伤还没痊愈，新的疼痛就要来临。那么请问在 t 时刻小燕儿的手肿得有多高呢？那其实就是将 t 时刻肿胀的高度加上 t - 1 时刻愈合后还剩的高度，再加上 $t-2$ 时刻还剩的高度……如式（6.1）所示，在 t 时刻小燕儿手肿胀的高度就是一个卷积的过程。

$$\text{肿胀高度}(t)=\int_{-\infty}^{\infty}\text{打击力度}(a)\times\text{权重}(t-a)\text{d}a=\sum_{a=-\infty}^{\infty}\text{打击力度}(a)\times\text{权重}(t-a) \tag{6.1}$$

一般而言，我们会用式（6.2）所示形式，使用星号表示卷积的过程。

$$s(t)=(x*w)(t) \tag{6.2}$$

在机器学习中，我们的输入数据通常是一个多维数组，因此卷积核也将是一个多维数组。再看式（6.1），我们的积分或求和是从负无穷到正无穷的，也就是计算小燕儿手的肿胀高度也要从很久很久之前开始，但十年前的伤口早就好了，其权重也就是 0。更确切地说，手肿卷积只在小段时间内有效，超过该范围的影响因子都为零。这也就意味着，在实践中我们不需要进行无限的累加，只需要有限的累加计算即可。

假设使用二维图像 I 作为我们的输入，其二维卷积核用 K 表示，如式（6.3）所示，就为该输入图像的卷积。

$$\text{S}(i,j)=(I*K)(i,j)=\sum_m\sum_n I(m,n)K(i-m,j-\text{n}) \tag{6.3}$$

卷积适合交换律，因此还可以写成如式（6.4）所示的形式。

$$\text{S}(i,j)=(K*I)(i,j)=\sum_m\sum_n I(i-m,j-n)K(m,n) \tag{6.4}$$

通常式（6.4）更容易在机器学习库中实现。在上述公式中，卷积其实需要把卷积核进行翻转后再进行加权求和，也就是输入的索引增加，卷积核的索引应该减少。之所以翻转卷积核是为了满足卷积的交换律性质，但卷积交换律在神经网络实现中并不重要。因此在神经网络中，如式（6.5）所示，实现的其实是**互相关**（Cross-Correlation）操作，和卷积类似但不需要翻转卷积。

$$S(i,j)=(I*K)(i,j)=\sum_m\sum_n I(i+m,j+n)K(m,n) \tag{6.5}$$

在大多数机器学习库中，虽然实现的是互相关操作，但仍然称其为卷积，而我们也将蒙住自己的眼睛将**互相关操作继续称为卷积操作**。

如果看完上述内容觉得快要昏厥了，那就忘记上面没用的内容。只需要记住，在机器学习中卷积就是局部特征乘以对应的权重，然后再累加起来即可。回到我们开始时的例子，长官从左到右，从上到下拿着小手电筒扫描油画，这就是所谓的“卷积”。如图 6-1 所示，是没有进行卷积核翻转操作情况下的卷积操作，并且在该例子中，卷积核权重与数据局部特征一一对应，此时也叫作**有效卷积**（Valid Convolution），详细信息可参看 6.4.2 节。

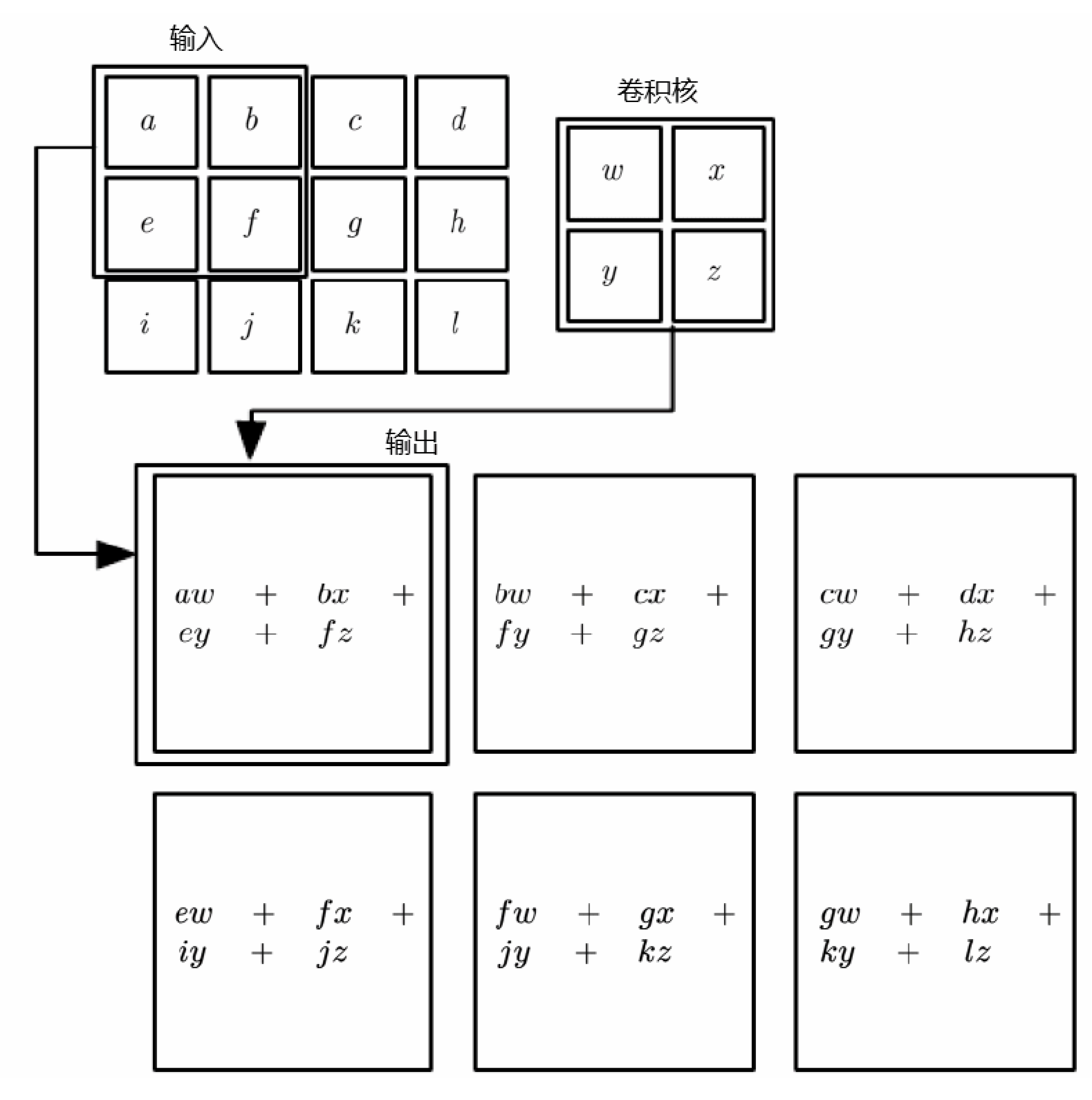


图 6-1　进行图像卷积示意图

6.2　卷积的意义

先讲一则故事，1958 年的某天，小猫咪小花收到好友哈贝（Hubel）和威塞勒（Wiesel）的邀请，到他们的实验室喝茶。在愉快聊天时，小花得知他俩正在研究瞳孔区域与大脑皮层神经元的对应关系，可由于缺少志愿者还在一筹莫展。而就在此时，小花心中那神圣的科学献身精神突然就爆发了，为了朋友，也为了科学，小花愿意作为志愿者帮助他俩。后来，他们就在小花的眼前，展示各种形状、各种亮度的物体，并且在展示每一件物体时，改变物体放置的位置和角度。他们通过观察发现，小花视觉系统中特定的神经元只会对一些特定形状敏感，并且移动这些形状的位置，特定的神经元依旧可以激活。此后他们经过多年的研究，

最终发现哺乳动物视觉系统初级视皮层（Primary Visual Cortex，V1）的秘密。上述故事是作者对 Hubel 和 Wiesel 两位伟大神经生物学家对视觉皮层早期研究的童真演义，现实的研究当然是充满了无趣和残忍（无数猫咪的后脑勺被插入电极），就如电影《少年派的奇幻漂流》结尾说的那样，你更喜欢哪种故事呢？为了表彰他们对视觉系统中信息加工所做出的重大贡献，1981 年他们共同分享了当年的诺贝尔生理或医学奖。

我们就简单地说一下 V1 视皮层具有的三个重要性质。

1. V1 层就如一张网一样排列在空间中，当光线仅穿过视网膜的下半部分时，V1 对应的一半区域就进入兴奋。

2. V1 包含着许多简单细胞，这些简单细胞仅对图像中小部分区域进行线性映射，这也称为**局部感受野**（Localized Receptive Field）[2]。而卷积网络的卷积特征提取单元也主要仿真简单细胞的这一性质。

3. V1 也包含着许多复杂细胞，它们在简单细胞中探测特征，并且对于特征的小幅平移具有不变性的检测能力，这也是卷积网络中**池化**（Pooling）单元的灵感来源。同时这些复杂细胞对于照明中的一些变化也具有不变能力，不能简单地通过在空间位置上池化来刻画，而这些不变性给卷积网络中的一些跨通道池化策略带来了灵感，例如 Maxout[3]单元。

如果从机器学习的角度出发，卷积带来了两个重要的思想：**稀疏连接**（Sparse Connectivity）及**参数共享**（Parameter Sharing）。这些内容我们早在第 4 章深度学习正则化章节就已经提到过了，接下来我们将详细地介绍这些思想。

6.2.1 稀疏连接

还是神奇手电筒照油画的例子，长官用的是小手电筒顺序地照油画，那你也许就会问了，如果换大一些的手电筒，一次性把整幅油画照出来又如何呢？这样的想法很好，关于大手电筒和小手电筒，就是我们本小节中将要讨论的全连接网络与局部连接或稀疏连接的内容。

如图 6-2（b）所示，在传统的神经网络中，每个神经元都会连接到上层的所有神经元中，这种连接也叫作**全连接**（Full Connectivity）[4]，而如图 6-2（a）所示，神经元只会连接到上层中的部分神经元，这种连接就被称为**稀疏连接**（Sparse Connectivity）[5]或**稀疏交互**（Sparse Interactions）。这两种连接最直观的区别是参数数量的巨大差异，参数数量是机器学习中模型能力强弱最直观的体现，模型能力又直接影响着是否容易过拟合，因此稀疏连接也算是一种有效防止过拟合的手段。

但稀疏连接也不仅仅是通过减小模型能力来提升泛化性能。例如在处理图像时，输入图片可能有数以百万计的像素，那么单独的像素和图像就几乎没有什么关联性，但我们可以通过在数百像素内探测小的有意义的**边角特征**来进一步处理图像。参数的减少也意味着我们所需的内存资源以及计算操作的减少，这对于效率的提升是巨大的。假设我们有 m 输入单元 n 输出单元，一次传播我们就需要 $m \times n$ 个参数连接。在实践中，每次计算就需要 $O(m \times n)$ 时间复杂度。如果我们将每个输出单元都限制在 k 连接参数的稀疏连接方式，那么所需的参数就减少为 $k \times n$ 个，计算时间复杂度就为 $O(k \times n)$。通常 k 往往远小于 m，因此这对于效率的提高是非常显著的。

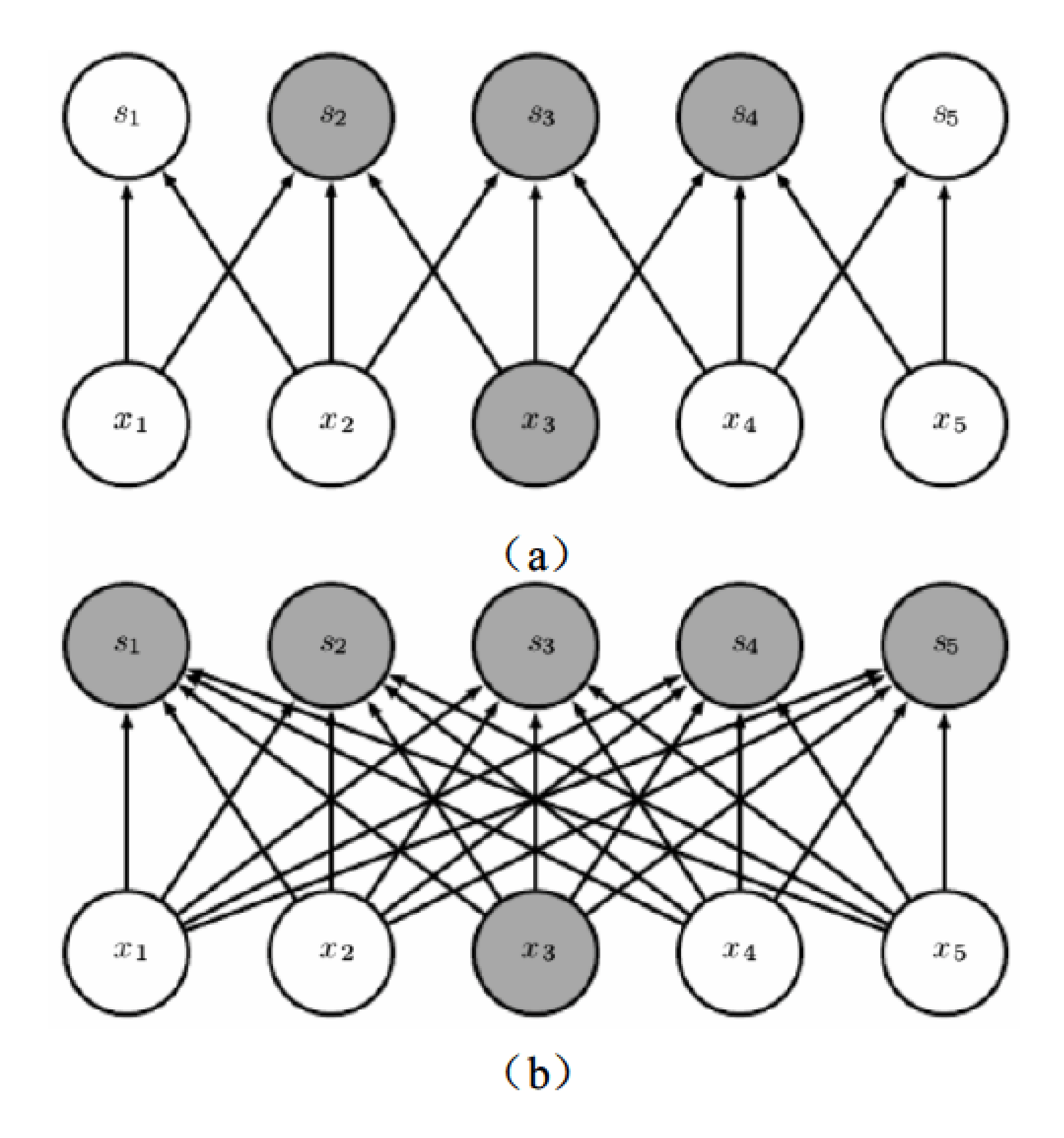


图 6-2　稀疏连接与全连接示意图

也许你会质疑，虽然参数大量地减少了，计算效率也提升了，但我们也丢失了大量的信息，这岂不是“自废双臂”吗？

确实，稀疏连接在单层中信息是不完整的，但这并没有严重到“自废双臂”的程度。在深层卷积网络中，稀疏网络会通过间接连接而补全低层网络的输入信息。如图 6-3 所示，虽然每个神经元都是稀疏连接，但 *g*3 仍然通过 *h*2、*h*3 和 *h*4 补全了所有的输入信息，多层稀疏连接使得网络仅仅依靠构造局部稀疏连接，就能高效地表述复杂的网络交互。

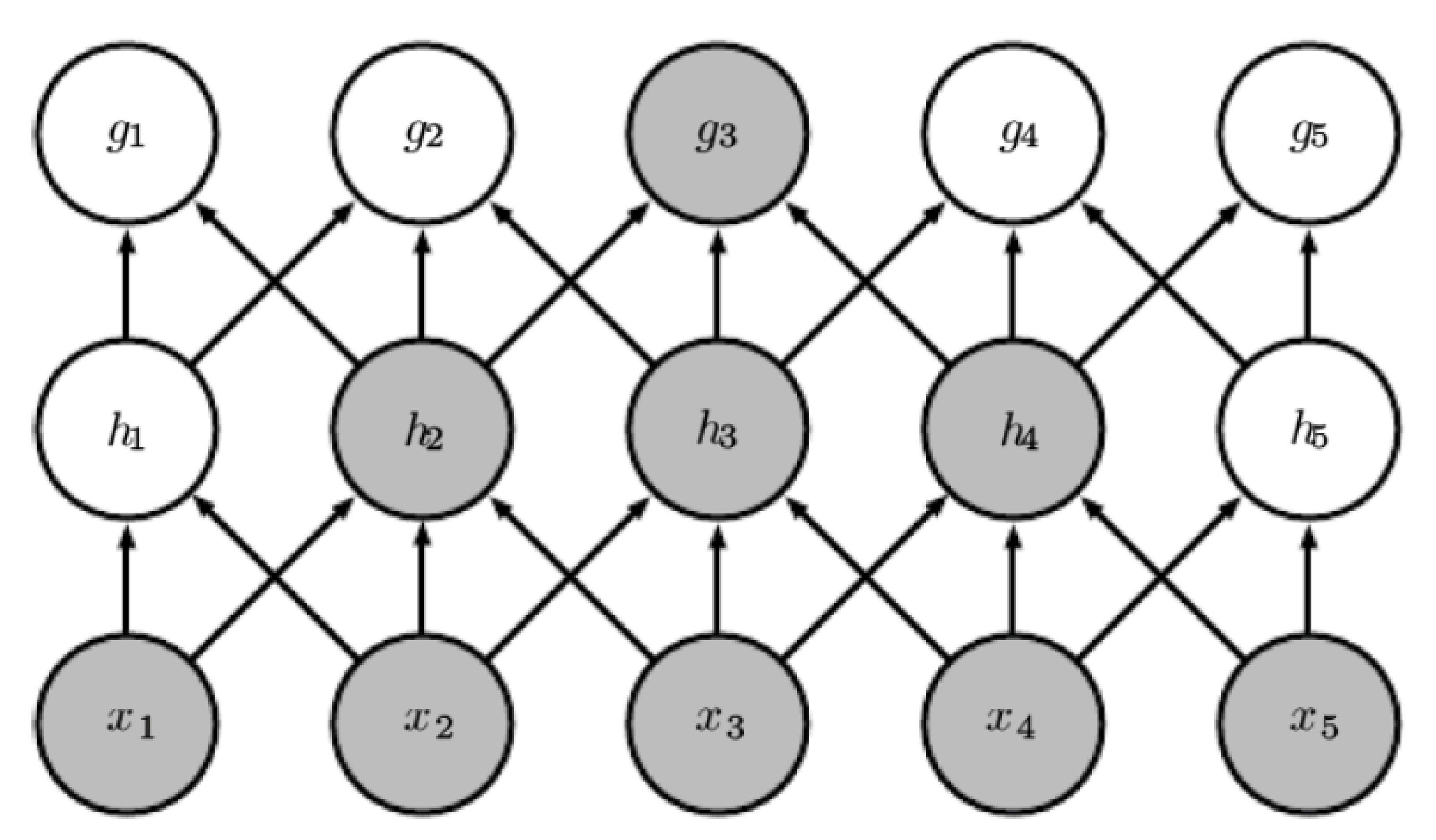


图 6-3　多层稀疏连接网络示意图

6.2.2　参数共享

参数共享（Parameter Sharing）指的是在模型的多处使用相同的参数。在上述例子中，使用小手电筒顺序照射的全过程也就是油画的每个位置都被同一把小手电照射，这就是所谓的参数共享。在神经网络中，参数共享还有一个同义词叫作**权重捆绑**（Tied Weights），这表示

在某处使用的权重，也将会被绑定到其他的地方使用。在卷积网络中，卷积核会在图像的各个位置进行卷积。因此卷积不是在特定的位置学习特征，而是在数据的各个区域提取特征。比如，某套卷积参数可以提取垂直边缘特征，该特征可能频繁存在图片的各个位置，因此经过卷积的扫面，实际上是对图片进行了一次垂直边缘特征过滤检测。

如图 6-4 所示，黑色箭头表示一条特定的连接权重，在图 6-4（a）中的卷积网络中，黑色箭头出现在每个输入神经元与对应的输出神经元中。而在图 6-4（b）中的全连接网络中，该黑色箭头仅仅出现在 *x*3 神经元到 *s*3 神经元中。

参数共享并没有影响神经网络的计算时间，其时间复杂度仍然为 $O(k \times n)$，但其显著地降低了需要存储的参数个数，原本需要存储 $k \times n$ 连接权重，而现在仅仅需要存储 k 个参数即可。比如识别一张 1000×1000 像素的图片，假设第一隐藏层神经元为 1 万个，那么在全连接网络中，仅仅这一层的参数就需要 100 亿个，这是个可怕的数字。假设我们使用卷积网络，其卷积核由 10×10 的一百个参数组成，如果是稀疏连接，参数就可以降低到 100 万个。假设我们再使用参数共享连接，那需要存储的参数就仅仅只有 100 个了，从 100 亿到 100 这是非常显著的降低。当然，在实践中我们会使用多个卷积核去提取多种特征，假设我们使用 100 个卷积核，那需要存储的参数也仅仅只有 1 万个，相比于 100 亿的数目，那也是如同进入天堂了一般。

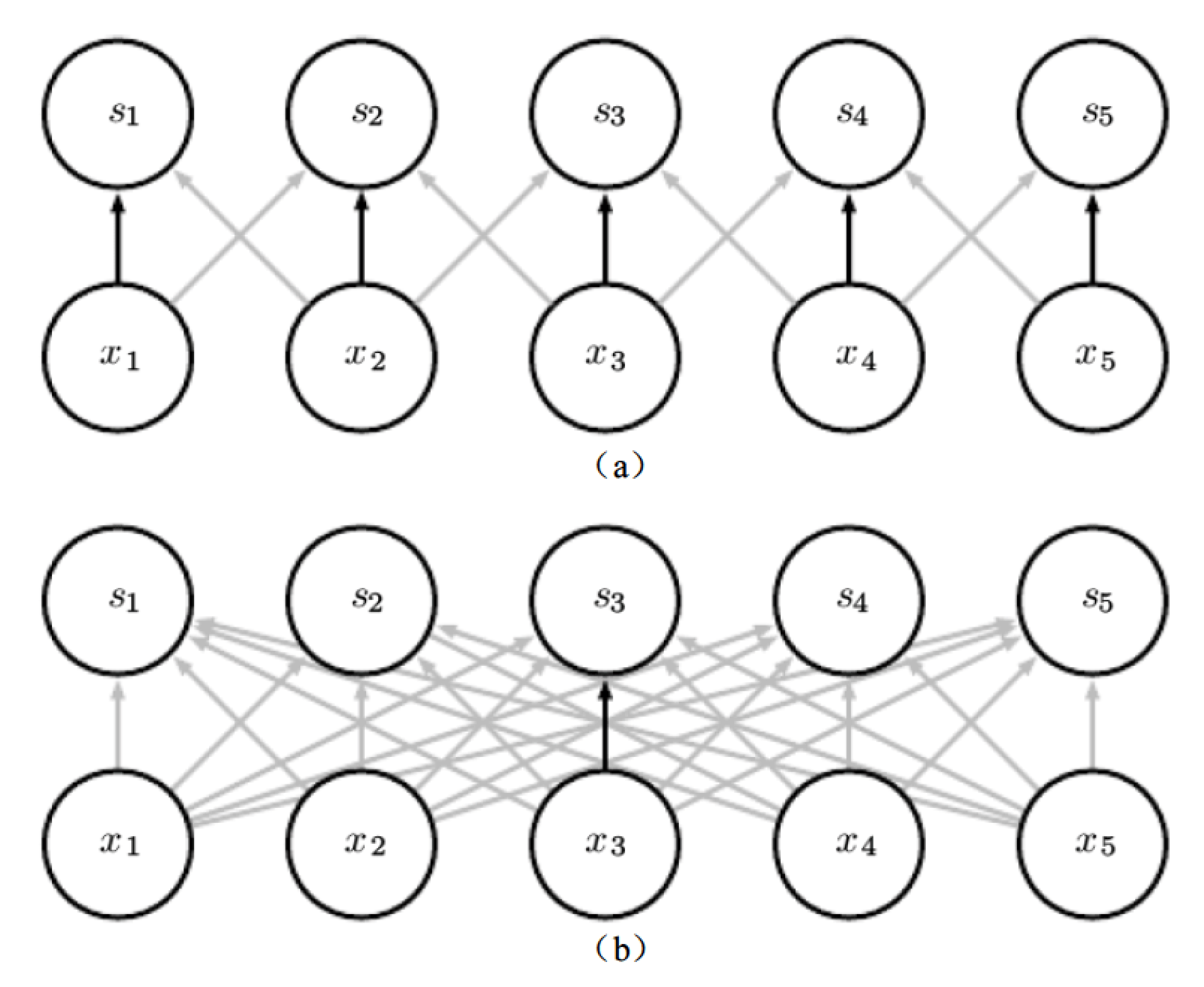


图 6-4　参数共享示意图

6.3　池化操作

在典型的卷积网络中，一层完整的卷积网络包含三个阶段：第一阶段称为**卷积层**，该层执行卷积操作生成一组特征图；第二阶段称为**探测层**，每一个特征值都会被送入一个非线性激活单元中进行激活；第三阶段称为**池化层**，负责将下层提取到的特征进行采样，缩小网络

规模。那么什么是池化呢？

池化操作非常简单，例如**最大池化**（Max Pooling）[6]其实就是筛选经过卷积映射后采样区域的最大值进行输出；**平均池化**（Average Pooling）[7]其实就是将经过卷积映射后采样区域的平均值进行输出。需要注意的是，池化操作不仅可以**缩小网络规模**，还能获取输入**数据的不变性特征**。

如果我们只在乎某些特征存不存在，而不在乎其存在的位置，那么局部不变性就是一个非常有用的属性。例如，识别图像中是否有人脸，我们不需要精确地知道眼睛的位置，只需要知道有一只眼睛在人脸的左边，有一只眼睛在人脸的右边即可。由于模糊位置加强了网络抗噪声的能力，因此提高了模型的泛化性能。在神经网络中我们就希望能加入不变性这样的先验知识，以此加快网络的训练，并且提升网络性能。

- 平移不变性

如图 6-5（a）所示，采用最大池化操作将探测层中相邻三个单元的最大值输出到池化层，当我们按照如图 6-5（b）所示那样平移输入值时，池化层的输出结果几乎没有变化。因为最大池化操作仅仅对周围的最大特征值敏感，而不在乎精确的位置，因此这种不敏感性反而使网络获得了平移不变性的能力。

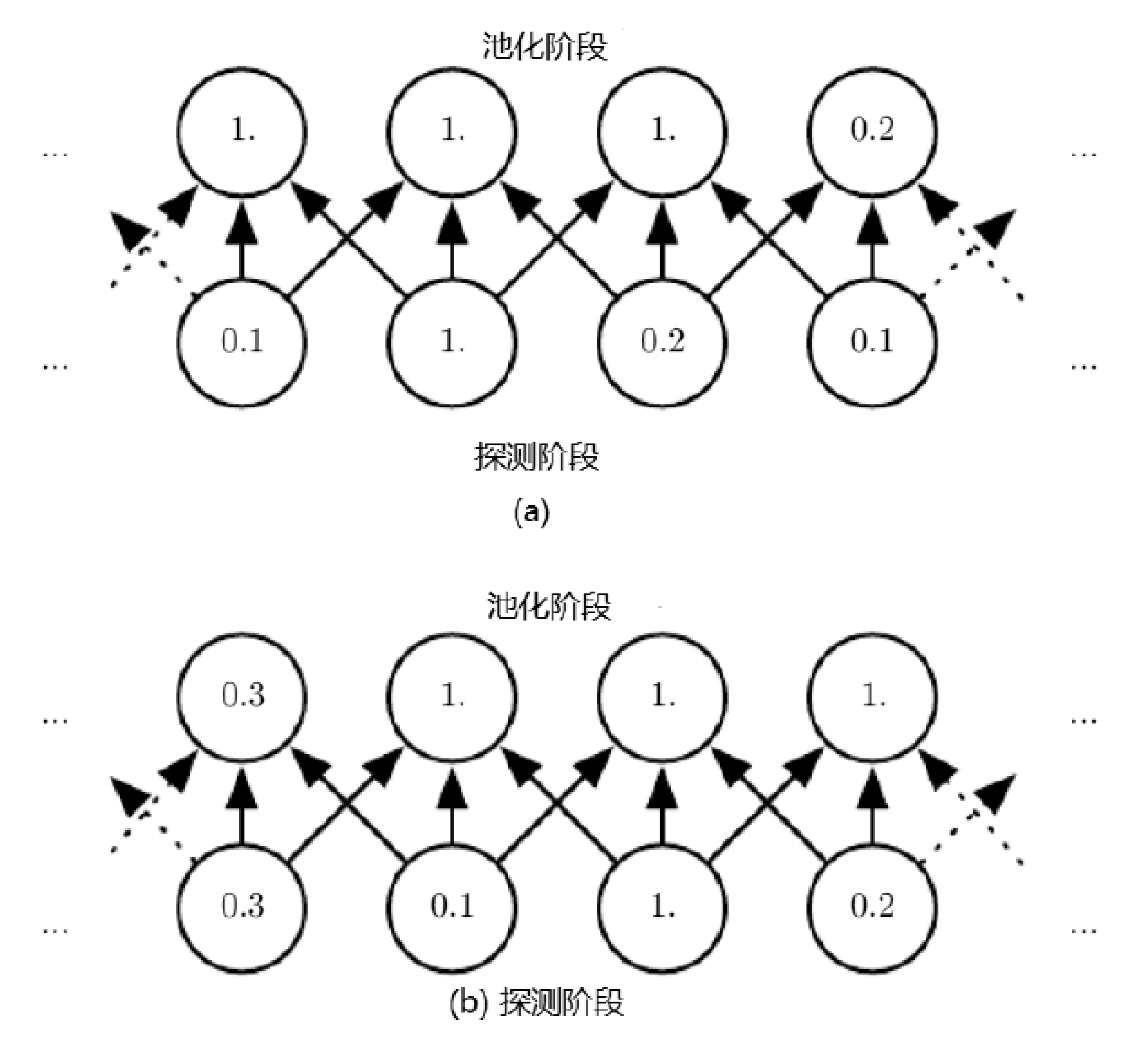


图 6-5　最大池化示意图

- 旋转不变性

上述内容中，我们将池化用于同一卷积探测层之后，让模型拥有了**平移不变性**的能力。如果我们将池化用于不同的卷积探测层，也就是在油画的同一位置使用不同的手电筒照射，然后再将不同的卷积结果进行最大池化，那我们就可以得到**旋转不变性**。如图 6-6 所示，假设我们有三个卷积探测器，每个探测器可以提取不同方向的数字图片“5”。当数字“5”出现在输入中时，对应的神经元就可以被激活。如图 6-6（a）所示，输入一张向上旋转的数字“5”，经过卷积检测后，第一个卷积单元就将被激活。如图 6-6（b）所示，输入一张向下旋

转的数字“5”，那么第三个卷积单元就将被激活。只要任意的卷积单元被激活，经过最大池化后，都可以识别出数字“5”。

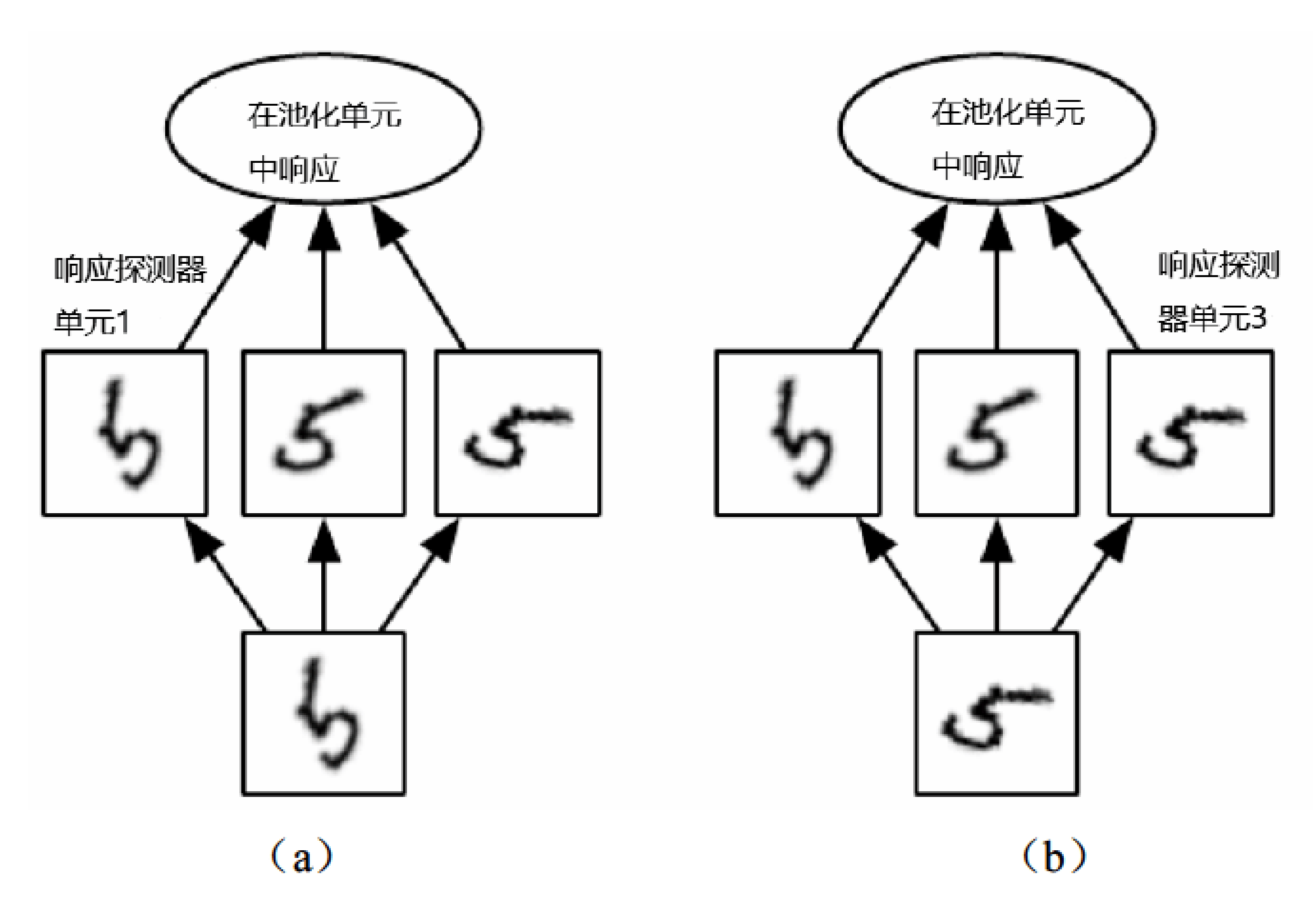


图 6-6 最大池化产生旋转不变性示意图

上述介绍的池化操作很像是一种另类的卷积操作，我们依然是在卷积层之后从左上角每隔一个单元池化一次，一直池化到右下角。但这种方式的计算消耗太大，我们只是选取周围节点的最大值或平均值，其实并不需要如此密集地进行池化。那我们可不可以跳跃着进行池化呢？比如 1，2，3 单元我们选取一个最大值，然后 4，5，6 单元再选取一个最大值，这就相当于池化操作跳跃了三个单元进行池化，而这种方式也被称之为**下采样**（DownSampling）。如图 6-7 所示，我们使用池化宽度为 3，跳跃两个单元进行采样。这种方式有效地提升了计算效率，假设 k 为跳跃的单元，那么在下一层中我们将降低大约 k 倍神经元输入数量。通过降低下一层维度，我们有效地减少下一层中的连接参数，不仅降低了内存消耗，而且还有利于学习效率的提升。

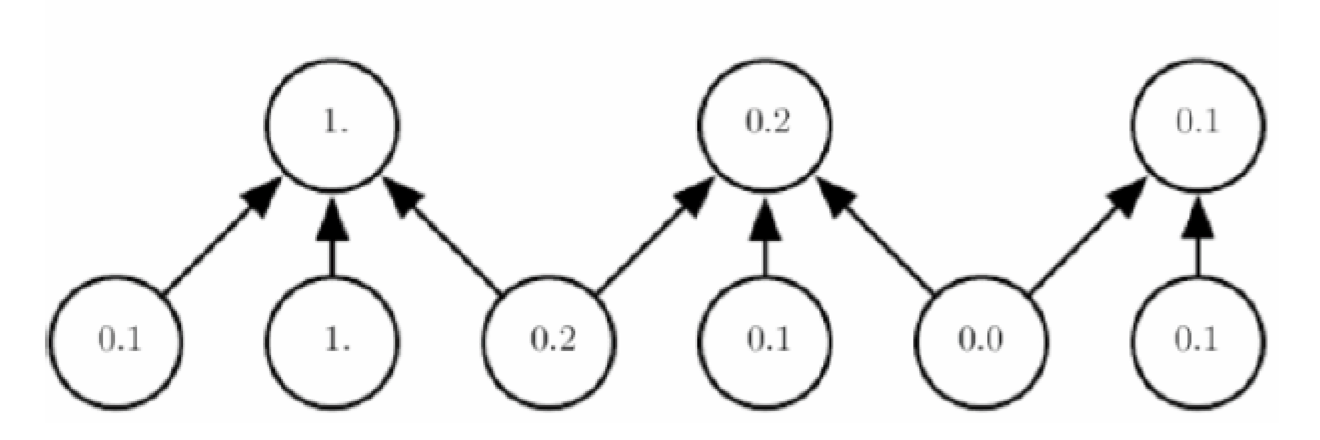


图 6-7　最大池化用于下采样示意图

池化还是**处理变长输入**的基本手段之一。例如，处理图像识别任务时所拥有的图片数据大小不相同，但是在神经网络中，网络是固定的，假设输入单元为 100，那我们只能输入 100 像素的图片。但如果拥有的是 110 像素的图片，那就需要对图片进行缩小或裁剪；如果拥有的是 90 像素的图片，那就需要对图片进行放大或填充；如果我们不想直接剪裁图片，那就可以利用池化操作将输入数据缩放在固定的尺寸上。

6.4 设计卷积神经网络

当在神经网络的环境中讨论卷积时，我们通常都不完全实现卷积操作。在实践中，我们通常都会根据需要对卷积网络进行轻微地修改，接下来我们就介绍一些重要的网络变种。

首先，神经网络中的卷积是由多个卷积核并行处理。由于特定的卷积核，在数据的多个位置进行卷积处理时，都只是在检测某一种特征。因此在网络的每一层，我们需要多个卷积核提取不同的卷积特征作为候选。

其次，我们的输入通常也不仅仅是格状的二维数据，也可能为三维数据。例如，在图像识别时，图片的每个像素都是由红、绿、蓝三种颜色构成。因此，输入数据就变成了三维数组，其中一维表示颜色不同的通道，另外两维表示每一通道中图像像素的二维坐标。

我们使用四维数组 $K_{i,j,k,l}$ 表示多道卷积核，其中下标 i 表示卷积核连接到输出的第 i 通道；下标 j 表示输入数据的第 j 通道；下标 k 表示输入数据的第 k 行；下标 l 表示输入数据的第 l 列。而我们使用数组 $V_{i,j,k}$ 表示第 i 道 j 行 k 列的输入数据，那输出单元 Z 的第 i 道 j 行 k 列的卷积结果就如式（6.6）所示。

$$Z_{i,j,k}=\sum_{l=1}^{L}\sum_{m=1}^{M}\sum_{n=1}^{N}V_{l,j+m-1,k+n-1}K_{i,l,m,n} \tag{6.6}$$

公式看起来可能有些复杂，但其实所表达的含义特别简单，其实就是将大小为 m 行 n 列的各通道数据，都乘以各自的权重，然后加起来放到对应的输出单元。由于数组下标是从 1 开始，因此需要减一。如果数据下标从零开始，可以直接将下标中的-1 去除。

6.4.1 跨步卷积

为了减少计算花费，在池化操作时我们选择**跨步**（stride）的方法，将卷积结果进行跨步池化（下采样）。同样地，在不显著影响特征提取的前提下，也可以使用**跨步卷积**的方式进行特征提取。我们也可以将跨步卷积看作是对卷积操作的输出结果进行下采样，如图 6-8（a）所示，我们使用步幅为 2 的跨步卷积提取特征，而在图 6-8（b）中我们使用步幅为 1 的默认跨步进行完整的特征提取，然后再在提取到的特征图中进行跨步为 2 的下采样操作。显然，这两种方式在效果上是等价的，但卷积之后采样明显更浪费计算机资源，因此在实际操作中我们会使用跨步卷积，而不考虑卷积之后再采样。

如果我们想要在输入数据的每个方向上进行采样，我们只需要在每次卷积之后，将行下标和列下标同时乘以 s 即可，如式（6.7）所示。当然，也可以根据实际需要分别对行和列，进行不同步幅的跨越。

$$Z_{i,j,k}=c(K,V,s)_{i,j,k}=\sum_{l=1}^{L}\sum_{m=1}^{M}\sum_{n=1}^{N}V_{l,(j-1)\times s+m,(k-1)\times s+n}K_{i,l,m,n} \tag{6.7}$$

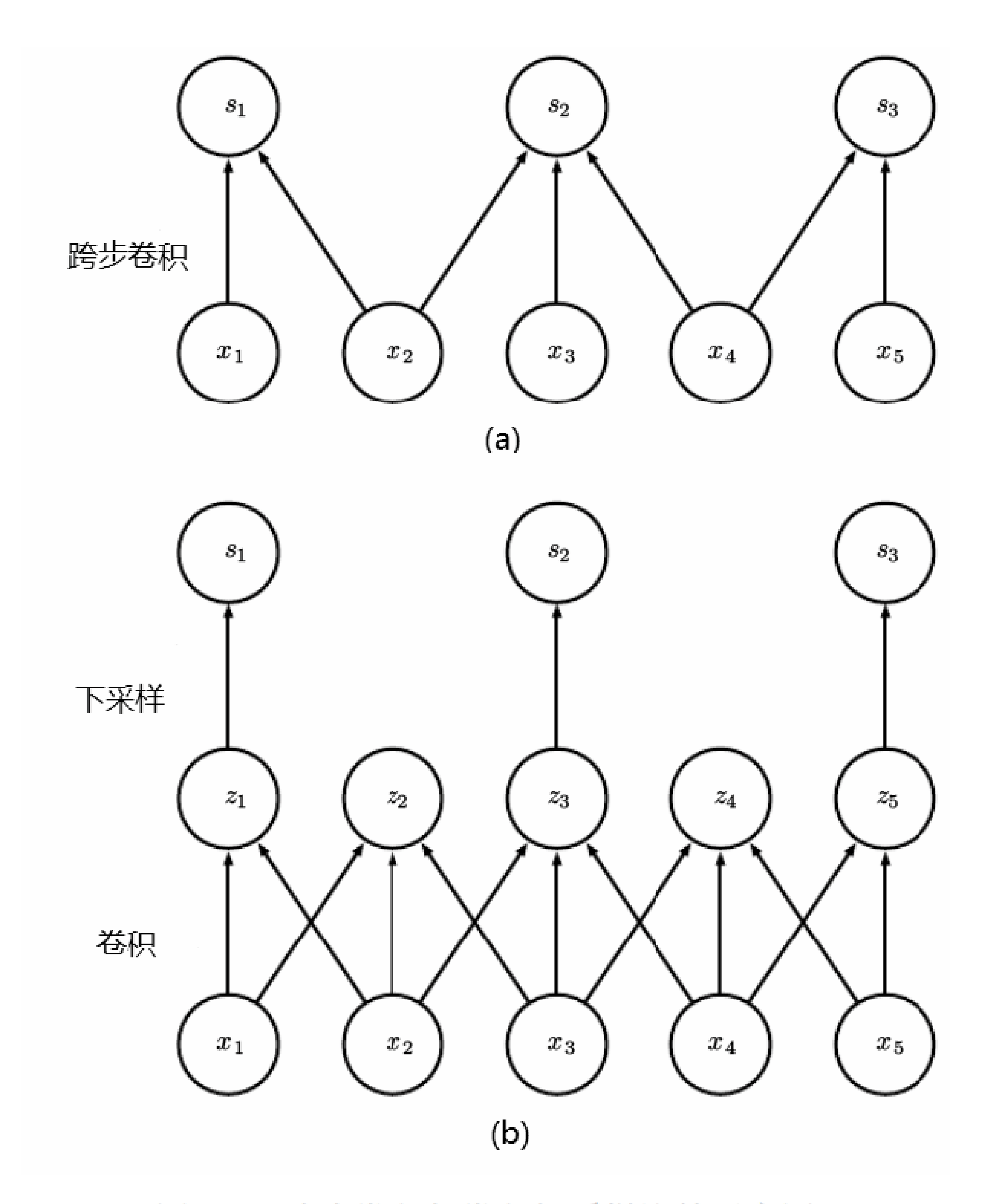


图 6-8　跨步卷积与卷积加采样比较示意图

6.4.2　零填充

在默认情况下，卷积网络每经过卷积核大小为 k 的卷积后，网络大小至少会缩小 k-1。如图 6-9 所示，假设我们的输入数据为 16 维，卷积核大小为 6，那么在默认情况下，我们的网络每层就会减少 5，因此该网络会在三层之后将网络输出缩减为 1。我们之所以将“神经网络”改名为“深度学习”，就是想强调我们应该使用深层的网络模型去表征数据。但在默认情况下，我们想要深层的网络模型就需要减小卷积核的大小，但减小卷积核大小，特征提取的能力也将减弱。因此，这是一个两难的问题。

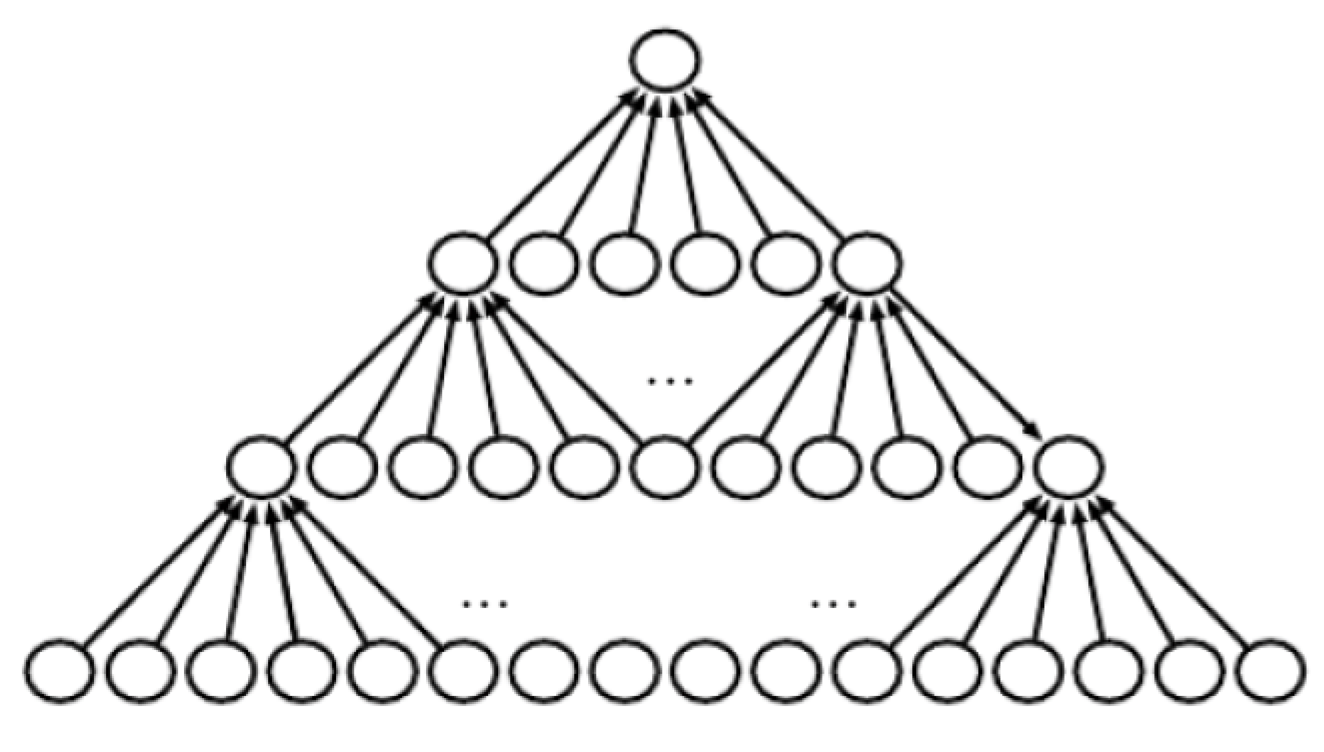


图 6-9　卷积网络收缩示意图

所谓**零填充**（Zero Padding），就是在每一层网络的边缘填充上输出为零的神经元。如图6-10所示，我们在每一层网络的边缘都加入5个输出为零的单元（黑色实心节点），那就刚好抵消使用卷积核为6的网络缩减的影响。我们就可以任意地选择卷积网络的层数，而不需要考虑卷积缩减的影响。



图 6-10　零填充卷积网络示意图

- **有效卷积**（Valid Convolution）：有效卷积其实就是没有零填充的卷积，该卷积要求卷积核与输入单元一一对应，图6-9就可以看作是一个有效卷积的网络结构示意图。假设输入单元的宽度为 m，卷积核的宽度为 k，那其输出的宽度就为 $m - k + 1$。随着网络层数的增加，该网络每一层的神经元数量也会显著地减少，直至缩减为1。因此，有效卷积需要仔细衡量卷积核尺寸与网络层数的利弊。
- **相同卷积**（Same Convolution）：相同卷积就是填充零神经元将网络补充回原来大小的卷积操作。如图6-10就可以看作是一个相同卷积的网络结构示意图，由于该卷积没有改变网络结构，因此可以自由地选择网络的层数及卷积核的尺寸。但由于网络边缘实际连接参数较少，在网络的边缘会出现欠表示现象。
- **全卷积**（Full Convolution）：全卷积则是最极端的一种零填充的方式，经过全卷积后，神经元的数量不但不会减少，还会增加。假设输入单元的宽度为 m，卷积核的宽度为 k，那么输出的宽度就为 $m + k - 1$，该过程在网络左右边缘各添加 $k - 1$ 个零神经元进行卷积。假如卷积核尺寸为6，第一个卷积提取神经元，会将第一个输入单元与卷积核第6个连接权重进行相乘；第二个卷积提取单元，会将第1-2个输入单元与卷积核第5-6个连接权重进行相乘相加；到最后一个卷积提取神经元时，就将最后一个输入单元与卷积核第1个连接权重进行相乘。这种卷积方式在网络边缘会出现更为严重的欠表示现象。

在实践中，最佳的零填充数量总是介于有效卷积与相同卷积之间。

6.4.3　非共享卷积

在某些情况下，我们并不一定真正想用卷积，而只是构造局部连接[8]的网络层。这种网络结构和卷积网络相同，只是每一个“局部卷积核”都有着不同且独立的连接权重，这种局部连接网络我们也称之为**非共享卷积**（Unshared Convolution）。如式（6.8）所示，我们可以

使用一个 6 维数组进行表示。

$$Z_{i,j,k} = \sum_{l}\sum_{m}\sum_{n} V_{l,j+m-1,k+n-1} w_{i,j,k,l,m,n} \qquad (6.8)$$

其中 $w_{i,j,k,l,m,n}$ 表示非共享卷积权矩阵，而 i 表示输出的通道，j 表示输出的行，k 表示输出列，l 表示输入的通道，m 表示输入的行，n 表示输入列。

当数据具有空间局部特征，但该特征并没有重复出现在整个特征空间时，局部连接层就会变得很有用。如图 6-11 所示，比较了卷积连接、局部连接和全连接的网络结构示意图，其中不同字母表示不同的连接权重。在局部连接中，每个连接权重都是不同的，而在卷积连接中，连接权重多次重复出现。

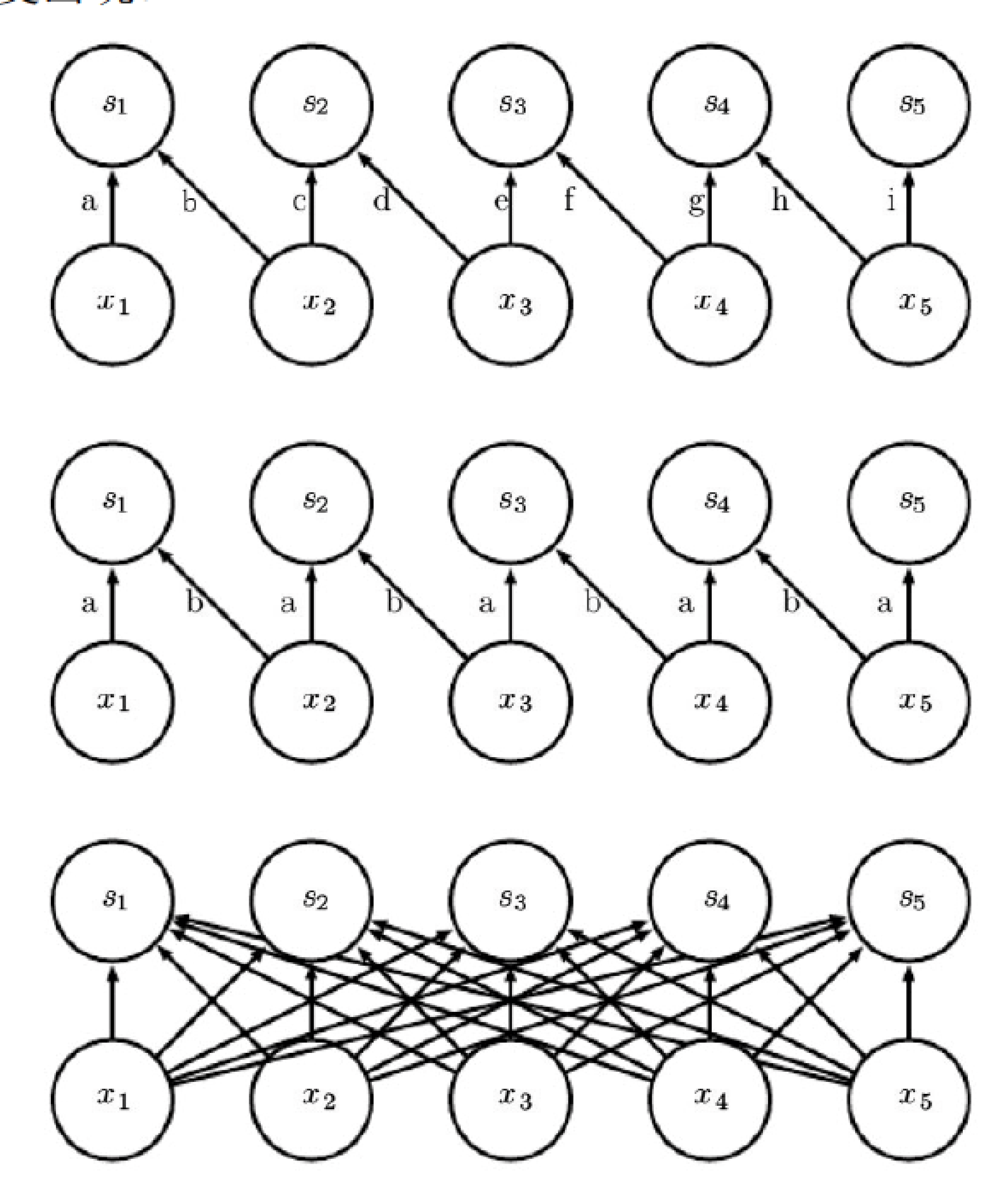


图 6-11　局部连接、卷积连接和全连接结构比较示意图

6.4.4　平铺卷积

我们将卷积连接想象成使用小手电筒在一幅油画上顺序地扫描照射，将局部连接想象成在油画的不同的位置用不同的小手电筒照射。各自的优缺点也就比较清晰了：一把手电筒（一套参数）始终能力有限，但在各个区域都使用不同的手电筒又略显“浪费”。我们能否做一个折中呢？比如捆绑两把或三把手电筒顺序地扫描？**平铺卷积**（Tiled Convolution）[9]就是介于卷积与局部连接之间的一种折中处理。

平铺卷积非常类似于同时使用多个跨步卷积**交替地**进行卷积处理。假设我们使用两个尺寸为 6 的卷积核进行平铺卷积，那么第一个卷积核函数就会与输入的第 1-6 单元进行卷积；第二个卷积核函数就会与第 2-7 单元进行卷积；然后第一个卷积核再与输入的第 3-8 单元卷积，

这样依次交替进行。如图 6-12 所示，比较了卷积、平铺卷积和局部连接的网络结构。我们使用 t 表示平铺卷积中卷积核的数量，如果 $t=1$ 就退化为传统的卷积操作；m 表示输入维度，k 表示卷积核大小，当 $t=m-k+1$ 时就退化为传统的局部连接。如式（6.9）所示，下标 i、j、k 分别表示第 i 输出通道 j 行 k 列，下标 l、m、n 表示第 l 输入通道 m 行 n 列，t 表示卷积核数量。而%表示取模运算，比如 $t\%t=0$；$(t+1)\%t=1$；$1\%t=1$。

$$Z_{i,j,k}=\sum_l\sum_m\sum_n V_{l,j+m-1,k+n-1}w_{i,l,m,n,j\%t+1,k\%t+1} \tag{6.9}$$

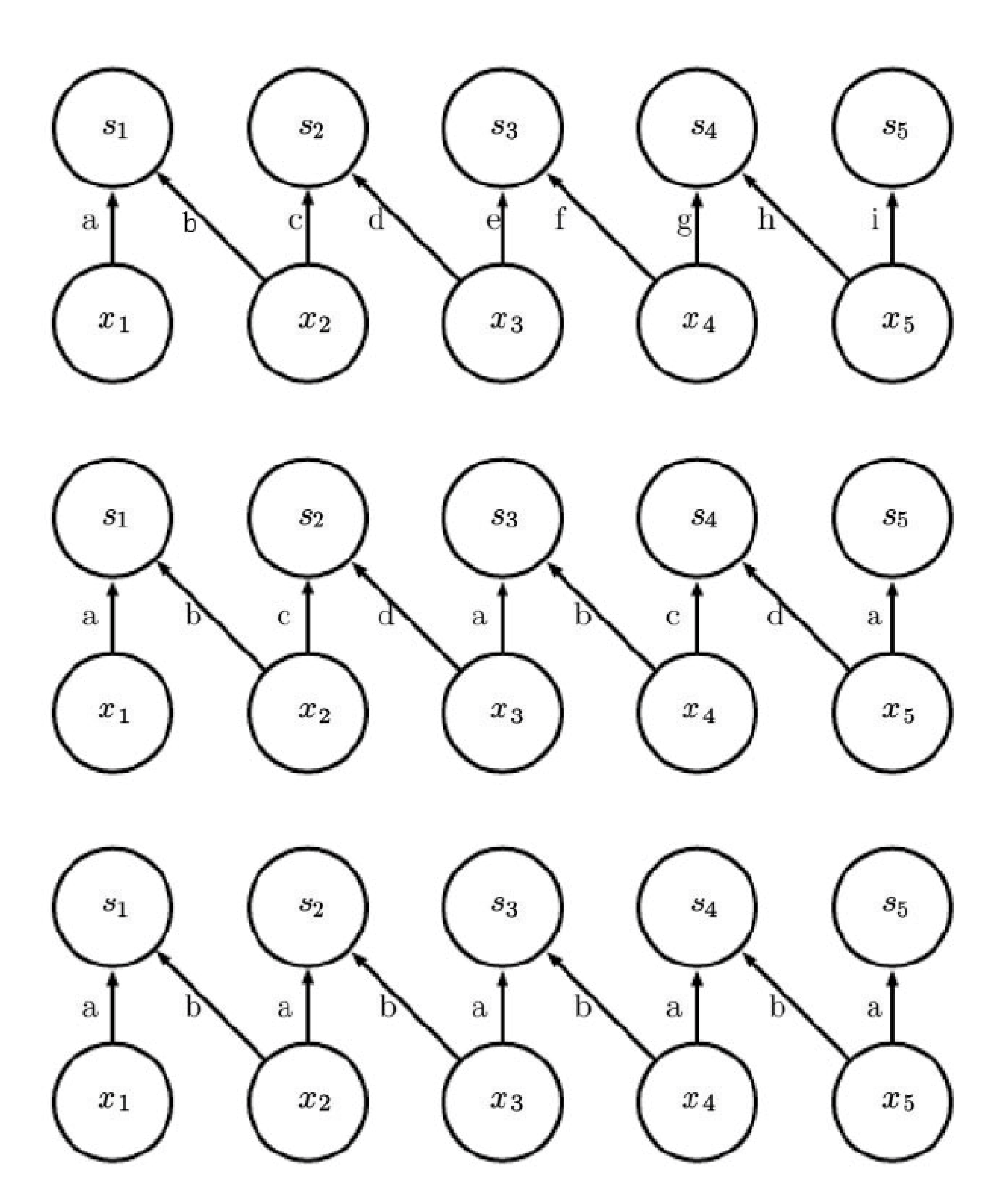


图 6-12　局部连接、平铺卷积和卷积结构比较示意图

6.5　卷积网络编码练习

在本章节的练习中，我们将对卷积、池化、空间批量归一化等内容进行编码，并比较不同实现方式的执行效率。由于 Python 语言运行效率太过缓慢，我们会在第 8 章 TensorFlow 快速入门章节中再次训练卷积网络，本章我们只需要针对卷积网络的各个模块进行编码练习即可。接下来，打开“第 6 章练习-卷积神经网络.ipynb”文件，进入本章练习。本章我们将逐步完成以下操作。

- 卷积前向传播编码练习；
- 卷积反向传播编码练习；
- 最大池化前向传播编码练习；

- 最大池化反向传播编码练习；
- 组合完整卷积层编码练习；
- 空间批量归一化编码练习。

首先是我们已经非常熟悉的库文件导入以及训练数据导入代码模块，本章的练习文件存放在“DLAction/classifiers/chapter6”目录中。

库文件，数据导入代码块：

```
#-*- coding: utf-8 -*-
import time
import numpy as np
import matplotlib.pyplot as plt
from classifiers.chapter6 import *
from utils import *
%matplotlib inline
plt.rcParams[ 'figure.figsize' ] = ( 10.0, 8.0 )
plt.rcParams[ 'image.interpolation' ] = 'nearest'
plt.rcParams[ 'image.cmap' ] = 'gray'
%load_ext autoreload
%autoreload 2
def rel_error( x, y ) :
  """ 相对误差 """
  return np.max( np.abs( x – y ) / ( np.maximum( 1e-8, np.abs( x ) + np.abs( y ) ) ) )
# 导入数据。
data = get_CIFAR10_data( )
for k, v in data.iteritems( ):
  print '%s: ' % k, v.shape
```

6.5.1　卷积前向传播

卷积的前向传播过程，其实就是使用多个卷积核顺序扫描输入数据的过程。虽然思想简单，但对于实际编程而言可能就会比较有难度了。我们已经实现了前向传播最直观的版本，但执行效率十分低下，可读性也非常差。该版本涉及很多个显式的循环，仔细阅读该版本代码并充分理解前向传播后，打开“DLAction/classifiers/chapter6/cnn_layers.py”文件，完成 conv_forward_naive 函数的编码练习，使用的显式循环越少，其执行效率就越高。

卷积前向传播代码块（多循环）：

```
def conv_forward_naive1( x, w, b, conv_param ) :
  """
  卷积层传播的慢速版本。
  该过程尽可能地逻辑清晰即可，你可以使用多个嵌套循环完成该函数。
```

```
Input:
- x：四维图片数据( N, C, H, W )，分别表示(数量，色道，高，宽)。
- w：四维卷积核( F, C, HH, WW )，分别表示(下层色道，上层色道，高，宽)。
- b：偏置项( F, )
- conv_param：字典型参数表，其键值为：
  - 'stride':跳跃数据进行卷积的跨幅数量。
  - 'pad':输入数据的零填充数量。
Returns 元组型:
- out：输出数据( N, F, H', W' ) ，其中 H' 和 W' 分别为：
  H' = 1 + ( H + 2 * pad – HH ) / stride
  W' = 1 + ( W + 2 * pad – WW ) / stride
- cache: ( x, w, b, conv_param )
"""
out = None
############################################################################
#                 任务：实现卷积层的前向传播。                              #
#            提示：可以使用 np.pad 函数进行零填充。                         #
############################################################################
N, C, H, W = x.shape[ 0 ], x.shape[ 1 ], x.shape[ 2 ], x.shape[ 3 ]
F, HH, WW = w.shape[ 0 ], w.shape[ 2 ], w.shape[ 3 ]
pad = conv_param[ 'pad' ]
stride = conv_param[ 'stride' ]
x_pad = np.pad( x, ( ( 0, ), ( 0, ), ( pad, ), ( pad, ) ), 'constant' )
Hhat = 1 + ( H + 2 * pad – HH ) / stride
What= 1 + ( W + 2 * pad – WW ) / stride
out = np.zeros( [ N, F, Hhat, What ] )
for n in xrange( N ) :
  for f in xrange( F ) :
      for i in xrange( Hhat ) :
          for j in xrange( What ) :
              xx = x_pad[ n, : , i * stride : I * stride + HH, j * stride : j* stride + WW ]
              out[ n, f, i, j ] = np.sum( xx * w[ f ] ) + b[ f ]
############################################################################
#                              结束编码                                    #
############################################################################
cache = ( x, w, b, conv_param )
return out, cache
```

阅读上述代码块后，实现下列代码块。

卷积前向传播代码块：

```
def conv_forward_naive( x, w, b, conv_param ) :
  """
  卷积前向传播。
  该过程尽可能地逻辑清晰即可，可以使用多个嵌套循环完成该函数。
  Input:
  - x：四维图片数据( N, C, H, W )，分别表示(数量，色道，高，宽)。
  - w：四维卷积核( F, C, HH, WW )，分别表示(下层色道，上层色道，高，宽)。
  - b：偏置项( F, )。
  - conv_param：字典型参数表，其键值为：
    - 'stride':跳跃数据进行卷积的跨幅数量。
    - 'pad':输入数据的零填充数量。
  Returns 元组型:
  - out：输出数据( N, F, H', W' ) ，其中 H' 和 W' 分别为：
    H' = 1 + ( H + 2 * pad – HH ) / stride
    W' = 1 + ( W + 2 * pad – WW ) / stride
  - cache: ( x, w, b, conv_param )
  """
  out = None
  ##################################################################
  #                    任务：实现卷积层的前向传播。                     #
  #                提示：可以使用 np.pad 函数进行零填充。                #
  ##################################################################




  ##################################################################
  #                             结束编码                              #
  ##################################################################
  cache = ( x, w, b, conv_param )
  return out, cache
```

完成上述代码块后，执行下列代码进行检验。

检验卷积前向传播代码块：

```
x_shape = ( 2, 3, 4, 4 )
w_shape = ( 3, 3, 4, 4 )
```

```
x = np.linspace( -0.1, 0.5, num = np.prod( x_shape ) ).reshape( x_shape )
w = np.linspace( -0.2, 0.3, num = np.prod( w_shape ) ).reshape( w_shape )
b = np.linspace( -0.1, 0.2, num = 3 )
conv_param = { 'stride': 2, 'pad': 1 }
out, _ = conv_forward_naive( x, w, b, conv_param )
correct_out = np.array( [ [ [ [ [ -0.08759809, -0.10987781 ],
                              [ -0.18387192, -0.2109216 ] ],
                            [ [ 0.21027089,   0.21661097 ],
                              [ 0.22847626,   0.23004637 ] ],
                            [ [ 0.50813986,   0.54309974 ],
                              [ 0.64082444,   0.67101435 ] ] ],
                           [ [ [ -0.98053589, -1.03143541 ],
                              [ -1.19128892, -1.24695841 ] ],
                            [ [ 0.69108355,   0.66880383 ],
                              [ 0.59480972,   0.56776003 ] ],
                            [ [ 2.36270298,   2.36904306 ],
                              [ 2.38090835,   2.38247847 ] ] ] ] ] )
# 误差应该在 1e-8 左右。
print '测试 conv_forward_naive 函数'
print '误差: ', rel_error( out, correct_out )
```

正确编码检验结果：
测试 conv_forward_naive 函数 误差: 2.21214764175e-08

6.5.2 卷积反向传播

对于初学者而言，卷积的反向传播会非常难以实现，如果使用显式循环的方式，可能会需要 6 层左右的循环。阅读 conv_backward_naive1 函数代码，当你理解该过程之后，打开“DLAction/classifiers/ chapter6/cnn_layers.py”文件，试着完成 conv_backward_naive 函数，并尽可能减少显式循环的使用数量。

卷积反向传播显式循环版本：

```
def conv_backward_naive1(dout, cache):
  """
  卷积层反向传播显式循环版本。
  Inputs:
  - dout:上层梯度。
  - cache: 前向传播时的缓存元组 ( x, w, b, conv_param )。
  Returns 元组:
```

```
- dx:   x 梯度。
- dw:   w 梯度。
- db:   b 梯度。
"""
dx, dw, db = None, None, None
##############################################################
#                    任务 ：实现卷积层反向传播。                    #
##############################################################
x, w, b, conv_param = cache
P = conv_param[ 'pad' ]
x_pad = np.pad( x, ( ( 0, ), ( 0, ), ( P, ), ( P, ) ), 'constant' )
N, C, H, W = x.shape
F, C, HH, WW = w.shape
N, F, Hh, Hw = dout.shape
S = conv_param[ 'stride' ]
dw = np.zeros( ( F, C, HH, WW ) )
for fprime in range( F ) :
    for cprime in range( C ) :
        for i in range( HH ) :
            for j in range( WW ) :
                sub_xpad = x_pad[ : ,cprime, i : i + Hh * S : S, j : j + Hw * S : S ]
                dw[ fprime, cprime, i, j ] = np.sum( dout[ : , fprime , : , : ] * sub_xpad )
db = np.zeros( ( F ) )
for fprime in range( F ) :
    db[ fprime ] = np.sum( dout[ : , fprime, : , : ] )
dx = np.zeros( ( N, C, H, W ) )
for nprime in range( N ) :
    for i in range( H ) :
        for j in range( W ) :
            for f in range( F ) :
                for k in range( Hh ) :
                    for l in range( Hw ) :
                        mask1 = np.zeros_like( w[ f, : , : , : ] )
                        mask2 = np.zeros_like( w[ f, : , : , : ] )
                        if ( i + P – k * S ) < HH and ( i + P – k * S ) >= 0 :
                            mask1[ : , i + P – k * S, : ] = 1.0
                        if ( j + P – l * S ) < WW and ( j + P – l * S ) >= 0 :
                            mask2[ : , : , j + P – l * S ] = 1.0
                        w_masked = np.sum( w[ f, : , : , : ] * mask1 * mask2, axis = ( 1, 2 ) )
```

```
                        dx[ nprime, : , i , j ] + = dout[ nprime, f, k, l ] * w_masked
    #####################################################################
    #                               结束编码                            #
    #####################################################################
    return dx, dw, db
```

阅读多循环版本后，试着减少循环，完成下列代码块。

conv_backward_naïve 函数代码块：

```
def conv_backward_naive( dout, cache ):
    """
    卷积层反向传播。
    Inputs:
    - dout:上层梯度。
    - cache: 前向传播时的缓存元组（x, w, b, conv_param )。
    Returns 元组:
    - dx: x 梯度。
    - dw: w 梯度。
    - db: b 梯度。
    """
    dx, dw, db = None, None, None
    #################################################################
    #                   任务：实现卷积层反向传播。                    #
    #################################################################




    #################################################################
    #                           结束编码                              #
    #################################################################
    return dx, dw, db
```

完成上述代码块后，使用下列代码进行梯度检验。

conv_backward_naive 梯度检验代码块：

```
x = np.random.randn( 4, 3, 5, 5 )
w = np.random.randn( 2, 3, 3, 3 )
b = np.random.randn( 2, )
dout = np.random.randn( 4, 2, 5, 5 )
```

```
conv_param = { 'stride': 1, 'pad': 1 }
dx_num = eval_numerical_gradient_array( lambda x: conv_forward_naive( x, w, b, conv_param )[ 0 ], x, dout )
dw_num = eval_numerical_gradient_array( lambda w: conv_forward_naive( x, w, b, conv_param )[ 0 ], w, dout )
db_num = eval_numerical_gradient_array( lambda b: conv_forward_naive( x, w, b, conv_param )[ 0 ], b, dout )
out, cache = conv_forward_naive( x, w, b, conv_param )
dx, dw, db = conv_backward_naive( dout, cache )
# 相对错误大约为 1e-9'。
print '测试 conv_backward_naive 函数'
print 'dx 误差: ', rel_error( dx, dx_num )
print 'dw 误差: ', rel_error( dw, dw_num )
print 'db 误差: ', rel_error( db, db_num )
```

正确编码后的梯度检验结果：

```
测试 conv_backward_naive 函数
dx 误差:   4.19390482385e-09
dw 误差:   6.74985341202e-10
db 误差:   7.5566263037e-12
```

6.5.3 最大池化前向传播

同样地，我们也已经实现了 max_pool_forward_naive1 函数。仔细阅读该代码后，试着尽可能少地使用显式循环，完成最大池化的前向传播。打开“DLAction/classifiers/chapter6/cnn_layers.py”文件，完成 max_pool_forward_naive 函数，实现最大池化前向传播。

max_pool_forward_naive1 函数代码块：

```
def max_pool_forward_naive1(x, pool_param):
  """
  实现慢速版本的最大池化操作前向传播。
  Inputs:
  - x: 数据 ( N, C, H, W )
  - pool_param: 键值：
    - 'pool_height': 池化高度。
    - 'pool_width': 池化宽度。
    - 'stride': 步幅。
  Returns 元组型:
  - out: 输出数据。
  - cache: ( x, pool_param )
  """
  out = None
  ###########################################################################
```

```
#                           任务：实现最大池化操作的前向传播。              #
###########################################################################
N, C, H, W = x.shape
pool_h = pool_param[ 'pool_height' ]
pool_w = pool_param[ 'pool_width' ]
stride = pool_param[ 'stride' ]
wHat = ( W - pool_w ) / stride + 1
hHat = ( H - pool_h ) / stride + 1
out = np.zeros( ( N, C, hHat, wHat ) )
for n in xrange( N ) :
        for c in xrange( C ) :
            for w in xrange( wHat ) :
                for h in xrange( hHat ) :
                    out[ n, c, h, w ] = np.max( x[ n, c, h * stride : h * stride + pool_h,
                                                  w * stride : w * stride + pool_w ] )
###########################################################################
#                                  结束编码                              #
###########################################################################
cache = ( x, pool_param )
return out, cache
```

阅读上述代码后，完成下列代码块。

max_pool_forward_naive 函数代码块：

```
def max_pool_forward_naive( x, pool_param ) :
  """
  最大池化前向传播。
  Inputs:
  - x: 数据 （N, C, H, W )。
  - pool_param: 键值：
    - 'pool_height': 池化高度。
    - 'pool_width': 池化宽度。
    - 'stride': 步幅。
  Returns 元组型:
  - out: 输出数据。
  - cache: ( x, pool_param )
  """
  out = None
  ###########################################################################
  #                           任务：实现最大池化操作的前向传播。              #
```

```
##################################################################






##################################################################
#                              结束编码                            #
##################################################################
cache = ( x, pool_param )
return out, cache
```

完成上述代码后，使用下列代码检验最大池化前向传播。

max_pool_forward_naive 函数检验代码块：

```
x_shape = ( 2, 3, 4, 4 )
x = np.linspace( -0.3, 0.4, num = np.prod( x_shape ) ).reshape( x_shape )
pool_param = { 'pool_width': 2, 'pool_height': 2, 'stride': 2 }
out, _ = max_pool_forward_naive( x, pool_param )
correct_out = np.array ( [ [ [ [   -0.26315789, -0.24842105 ],
                              [ -0.20421053, -0.18947368 ] ],
                            [ [ -0.14526316, -0.13052632 ],
                              [ -0.08631579, -0.07157895 ] ],
                            [ [ -0.02736842, -0.01263158 ],
                              [ 0.03157895,   0.04631579 ] ] ],
                          [ [ [ 0.09052632,   0.10526316 ],
                              [ 0.14947368,   0.16421053 ] ],
                            [ [ 0.20842105,   0.22315789 ],
                              [ 0.26736842,   0.28210526 ] ],
                            [ [ 0.32631579,   0.34105263 ],
                              [ 0.38526316,   0.4          ] ] ] ] )
# 相对误差大约为 1e-8。
print '测试 max_pool_forward_naive 函数:'
print '误差: ', rel_error( out, correct_out )
```

正确编码后的检验结果：

```
测试 max_pool_forward_naive 函数:
误差:   4.16666651573e-08
```

6.5.4 最大池化反向传播

同样地，我们也已经实现了 max_pool_backward_naive1 函数，仔细阅读该代码后，试着尽可能少地使用显式循环，完成最大池化的反向传播。打开“DLAction/classifiers/chapter6/cnn_layers.py”文件，完成 max_pool_backward_naive 函数。

max_pool_backward_naive1 函数代码块：

```
def max_pool_backward_naive1( dout, cache ) :
  """
  最大池化反向传播显式循环版本。
  Inputs:
  - dout:  上层梯度。
  - cache:  缓存 （x, pool_param )。
  Returns :
  - dx:    x 梯度。
  """
  dx = None
  ###################################################################
  #                 任务：实现最大池化反向传播。                       #
  ###################################################################
  x, pool_param = cache
  Hp = pool_param[ 'pool_height' ]
  Wp = pool_param[ 'pool_width' ]
  S = pool_param[ 'stride' ]
  N, C, H, W = x.shape
  H1 = ( H – Hp ) / S + 1
  W1 = ( W – Wp ) / S + 1
  dx = np.zeros( ( N, C, H, W ) )
  for nprime in range( N ) :
        for cprime in range( C ) :
            for k in range( H1 ) :
                for l in range( W1 ) :
                    x_pooling = x[ nprime, cprime, k * S : k * S + Hp , l * S : l * S + Wp ]
                    maxi = np.max( x_pooling )
                    x_mask = x_pooling == maxi
                    dx[ nprime, cprime, k * S : k * S + Hp ,
                      l * S : l * S + Wp ] + = dout[ nprime, cprime , k, l ] * x_mask
  ####################################################################
  #                          结束编码                                 #
```

```
    ########################################################################
    return dx
```

阅读上述代码后，完成最大池化的反向传播。

max_pool_backward_naive 函数代码块：

```
def max_pool_backward_naive( dout, cache ) :
    """
    最大池化反向传播。
    Inputs :
    - dout : 上层梯度。
    - cache : 缓存 ( x, pool_param )。
    Returns :
    - dx :   x 梯度。
    """
    dx = None
    #########################################################################
    #                    任务：实现最大池化反向传播。                         #
    #########################################################################




    #########################################################################
    #                                结束编码                               #
    #########################################################################
    return dx
```

完成上述代码后，使用下列代码进行梯度检验。

最大池化反向传播梯度检验代码块：

```
x = np.random.randn( 3, 2, 8, 8 )
dout = np.random.randn( 3, 2, 4, 4 )
pool_param = { 'pool_height': 2, 'pool_width': 2, 'stride': 2 }
dx_num = eval_numerical_gradient_array(
    lambda x: max_pool_forward_naive( x, pool_param )[ 0 ], x, dout )
out, cache = max_pool_forward_naive( x, pool_param )
dx = max_pool_backward_naive( dout, cache )
# 相对误差大约为 1e-12。
print '测试 max_pool_backward_naive 函数:'
print 'dx 误差: ', rel_error( dx, dx_num )
```

正确编码后的检验结果：
测试 max_pool_backward_naive 函数: dx 误差: 3.27561726043e-12

6.5.5 向量化执行

由于 Python 自身执行速度缓慢的原因，即使我们完全向量化卷积后，该执行效率也远远达不到我们的需求。比较有效的方法是将完全向量版本的卷积操作使用 Cython 转化为 C 语言执行，但 Windows 下需要安装诸多依赖库，操作不太方便。因此我们会在第 8 章中使用 TensorFlow 在 GPU 环境下更加快速地执行卷积网络。此处只进行完全向量版本的执行，读者可简单地阅读下列代码，然后执行代码，比较各版本的效率，具体的训练卷积网络会在 8.4 章节中介绍。

- 快速卷积前向传播

接下来我们使用快速的卷积前向传播操作，该部分不做要求，直接使用即可。

conv_forward_fast 函数代码块：

```
def conv_forward_fast( x, w, b, conv_param ) :
  N, C, H, W = x.shape
  F, _, HH, WW = w.shape
  stride, pad = conv_param[ 'stride' ], conv_param[ 'pad' ]
  assert ( W + 2 * pad – WW ) % stride == 0, '宽度异常'
  assert ( H + 2 * pad – HH ) % stride == 0, '高度异常'
  p = pad   # 零填充。
  x_padded = np.pad( x, ( ( 0, 0 ), ( 0, 0 ), ( p, p ), ( p, p ) ), mode= 'constant' )
  # 计算输出维度。
  H + = 2 * pad
  W + = 2 * pad
  out_h = ( H – HH ) / stride + 1
  out_w = ( W – WW ) / stride + 1
  shape = ( C, HH, WW, N, out_h, out_w )
  strides = ( H * W, W, 1, C * H * W, stride * W, stride )
  strides = x.itemsize * np.array( strides )
  x_stride = np.lib.stride_tricks.as_strided( x_padded, shape = shape, strides = strides )
  x_cols = np.ascontiguousarray( x_stride )
  x_cols.shape = ( C * HH * WW, N * out_h * out_w )
  # 将所有卷积核重塑成一行。
  res = w.reshape( F, -1 ).dot( x_cols ) + b.reshape( -1, 1 )
  # 重塑输出。
```

```
    res.shape = ( F, N, out_h, out_w )
    out = res.transpose( 1, 0, 2, 3 )
    out = np.ascontiguousarray( out )
    cache = ( x, w, b, conv_param )
    return out, cache
```

阅读完上述代码后，执行下列代码块，比较快速卷积和多循环版本慢速卷积执行效率。

比较快速卷积和多循环版本慢速卷积执行效率的代码块：

```
from classifiers.chapter6.cnn_layers import conv_forward_fast
from time import time
x = np.random.randn( 100, 3, 31, 31 )
w = np.random.randn( 25, 3, 3, 3 )
b = np.random.randn( 25, )
dout = np.random.randn( 100, 25, 16, 16 )
conv_param = { 'stride': 2, 'pad': 1 }
t0 = time( )
out_naive, cache_naive = conv_forward_naive( x, w, b, conv_param )
t1 = time( )
out_fast, cache_fast = conv_forward_fast( x, w, b, conv_param )
t2 = time( )
print '测试 conv_forward_fast:'
print '慢速版本: %fs' % ( t1 - t0 )
print '快速版本: %fs' % ( t2 - t1 )
print '加速: %fx' % ( ( t1 - t0 ) / ( t2 - t1 ) )
print '误差: ', rel_error( out_naive, out_fast )
```

快速卷积与慢速卷积的执行结果：

```
测试 conv_forward_fast:
慢速版本: 0.188000s
快速版本: 0.024000s
加速: 7.833277x
误差:  1.66546731637e-10
```

- 快速池化操作

max_pool_forward_fast 函数代码块：

```
def max_pool_forward_fast( x, pool_param ) :
    N, C, H, W = x.shape
    pool_height = pool_param[ 'pool_height' ]
```

```
    pool_width = pool_param[ 'pool_width' ]
    stride = pool_param[ 'stride' ]
    assert pool_height == pool_width == stride, 'Invalid pool params'
    assert H % pool_height == 0
    assert W % pool_height == 0
    x_reshaped = x.reshape( N, C, H / pool_height, pool_height, W / pool_width, pool_width )
    out = x_reshaped.max( axis = 3 ).max( axis = 4 )
    cache = ( x, x_reshaped, out )
    return out, cache
```

max_pool_backward_fast 函数代码块：

```
def max_pool_backward_fast( dout, cache ):
    x, x_reshaped, out = cache
    dx_reshaped = np.zeros_like( x_reshaped )
    out_newaxis = out[ : , : , : , np.newaxis, : , np.newaxis ]
    mask = ( x_reshaped == out_newaxis )
    dout_newaxis = dout[ : , : , : , np.newaxis, : , np.newaxis ]
    dout_broadcast, _ = np.broadcast_arrays( dout_newaxis, dx_reshaped )
    dx_reshaped[ mask ] = dout_broadcast[ mask ]
    dx_reshaped / = np.sum( mask, axis = ( 3, 5 ), keepdims = True )
    dx = dx_reshaped.reshape( x.shape )
    return dx
```

阅读完上述代码后，执行下列代码，比较快速池化与多循环慢速池化的执行效率。

比较快速池化和多循环版本慢速池化执行效率的代码块：

```
x = np.random.randn( 100, 3, 32, 32 )
dout = np.random.randn( 100, 3, 16, 16 )
pool_param = { 'pool_height': 2, 'pool_width': 2, 'stride': 2 }
t0 = time( )
out_naive, cache_naive = max_pool_forward_naive( x, pool_param )
t1 = time( )
out_fast, cache_fast = max_pool_forward_fast( x, pool_param )
t2 = time( )
print '测试 pool_forward_fast:'
print '慢速版本: %fs' % ( t1 - t0 )
print '快速版本: %fs' % ( t2 - t1 )
print '加速: %fx' % ( ( t1 - t0 ) / ( t2 - t1 ) )
print '误差: ', rel_error( out_naive, out_fast )
t0 = time( )
```

```
dx_naive = max_pool_backward_naive( dout, cache_naive )
t1 = time( )
dx_fast = max_pool_backward_fast( dout, cache_fast )
t2 = time( )
print '\n 测试  pool_backward_fast:'
print '慢速版本: %fs' % ( t1 - t0 )
print '快速版本: %fs' % ( t2 - t1 )
print '加速: %fx' % ( ( t1 - t0 ) / ( t2 - t1 ) )
print 'dx  误差: ', rel_error( dx_naive, dx_fast )
```

比较池化前向传播结果：	比较池化反向传播结果：
测试 pool_forward_fast: 慢速版本: 0.007000s 快速版本: 0.003000s 加速: 2.333307x 误差:　0.0	测试 pool_backward_fast: 慢速版本: 0.018000s 快速版本: 0.014000s 加速: 1.285690x dx 误差:　0.0

6.5.6　组合完整卷积层

接下来我们将卷积、ReLU 和池化组合在一起，形成一层完整的卷积层。该部分内容和 3.4.3 组合单层神经元小节中介绍的组合 ReLU 层，组合 Dropout 层在逻辑结构上完全相同，请阅读 conv_relu_pool_forward 和 conv_relu_pool_backward 函数代码，然后进行测试。

conv_relu_pool_forward 函数代码块：

```
def conv_relu_pool_forward( x, w, b, conv_param, pool_param ) :
    a, conv_cache = conv_forward_fast( x, w, b, conv_param )
    s, relu_cache = relu_forward( a )
    out, pool_cache = max_pool_forward_fast( s, pool_param )
    cache = ( conv_cache, relu_cache, pool_cache )
    return out, cache
```

conv_relu_pool_backward 函数代码块：

```
def conv_relu_pool_backward( dout, cache ) :
    conv_cache, relu_cache, pool_cache = cache
    ds = max_pool_backward_fast( dout, pool_cache )
    da = relu_backward( ds, relu_cache )
    dx, dw, db = conv_backward_naive( da, conv_cache )
    return dx, dw, db
```

使用下列代码检验完整卷积层。

检验组合卷积层梯度代码块：

```
x = np.random.randn( 2, 3, 16, 16 )
w = np.random.randn( 3, 3, 3, 3 )
b = np.random.randn( 3, )
dout = np.random.randn( 2, 3, 8, 8 )
conv_param = { 'stride': 1, 'pad': 1 }
pool_param = { 'pool_height': 2, 'pool_width': 2, 'stride': 2 }
out, cache = conv_relu_pool_forward( x, w, b, conv_param, pool_param )
dx, dw, db = conv_relu_pool_backward( dout, cache )
dx_num = eval_numerical_gradient_array( lambda x: conv_relu_pool_forward(
            x, w, b, conv_param, pool_param )[ 0 ], x, dout )
dw_num = eval_numerical_gradient_array( lambda w: conv_relu_pool_forward(
            x, w, b, conv_param, pool_param )[ 0 ], w, dout )
db_num = eval_numerical_gradient_array( lambda b: conv_relu_pool_forward(
            x, w, b, conv_param, pool_param )[ 0 ], b, dout )
print '测试 conv_relu_pool'
print 'dx 误差: ', rel_error( dx_num, dx )
print 'dw 误差: ', rel_error( dw_num, dw )
print 'db 误差: ', rel_error( db_num, db )
```

梯度检验结果：

```
测试 conv_relu_pool
dx 误差:  2.79713149752e-07
dw 误差:  1.02470162321e-09
db 误差:  5.07494290358e-11
```

6.5.7 浅层卷积网络

接下来我们实现简单的浅层卷积网络，该网络由一层卷积层，两层全连接层组成：输入-conv-relu–(2 × 2)maxpool-affine-relu-affine–softmax 。打开“DLAction/classifiers/chapter6/cnn.py”文件，实现 ThreeLayerConvNet 类。

权重初始化代码块：

```
  def __init__( self, input_dim = ( 3, 32, 32 ), num_filters = 32, filter_size = 7,
                    hidden_dim = 100, num_classes = 10, weight_scale = 1e-3, reg = 0.0, ) :
    """
    初始化卷积网络。
    Inputs:
    - input_dim: 输入数据形状 ( C, H, W )。
```

```
    - num_filters: 卷积核数量。
    - filter_size: 卷积核尺寸。
    - hidden_dim: 全连接层隐藏层个数。
    - num_classes: 分类个数。
    - weight_scale: 权重规模（标准差）。
    - reg: 权重衰减因子。
    """
    self.params = { }
    self.reg = reg
    ############################################################################
    #                    任务：初始化权重参数。                                  #
    #  'W1'为卷积层参数，形状为( num_filters, C, filter_size, filter_size )。    #
    #  'W2'为卷积层到全连接层参数，形状为( ( H / 2 ) * ( W / 2 ) *              #
    #                                        num_filters,   hidden_dim )。    #
    #      'W3'隐藏层到全连接层参数。                                            #
    ############################################################################




    ############################################################################
    #                              结束编码                                    #
    ############################################################################
```

损失函数代码块：

```
def loss( self, X, y = None ) :
    W1, b1 = self.params[ 'W1' ], self.params[ 'b1' ]
    W2, b2 = self.params[ 'W2' ], self.params[ 'b2' ]
    W3, b3 = self.params[ 'W3' ], self.params[ 'b3' ]
    filter_size = W1.shape[ 2 ]
    conv_param = { 'stride': 1, 'pad': ( filter_size – 1 ) / 2 }
    pool_param = { 'pool_height': 2, 'pool_width': 2, 'stride': 2 }
    scores = None
    ####################################################################
    #                 任务：实现前向传播过程。                          #
    #            计算每类得分，将其存放在 scores 中。                   #
    ####################################################################
```

```
############################################################################
#                              结束编码                                      #
############################################################################
if y is None:
    return scores
loss, grads = 0, { }
############################################################################
#                          任务：实现反向转播。                               #
#                         注意：别忘了权重衰减项。                             #
############################################################################




############################################################################
#                              结束编码                                      #
############################################################################
return loss, grads
```

- 损失值检验

完成上述代码后，执行下列代码，进行简单的损失值检验。注意在不添加正则化的情况下，c 分类任务初始时的损失值应该接近于 $\log(c)$，运行下列代码。

检验三层卷积网络损失值：

```
model = ThreeLayerConvNet( )
N = 50
X = np.random.randn( N, 3, 32, 32 )
y = np.random.randint( 10, size = N )
loss, grads = model.loss( X, y )
print '初始损失值所对应分类 (无正则化): ', np.exp( loss )
model.reg = 0.5
loss, grads = model.loss( X, y )
print '初始损失值 (正则化): ', loss
```

正确编码后的检验结果：

```
初始损失值所对应分类 (无正则化):   9.99998941045
初始损失值 (正则化):   2.50850657265
```

- 梯度检验

接下来进行梯度检验，运行下列代码。

浅层卷积网络梯度检验：

```
num_inputs = 2
input_dim = ( 3, 16, 16 )
reg = 0.0
num_classes = 10
X = np.random.randn( num_inputs, *input_dim )
y = np.random.randint( num_classes, size = num_inputs )
model = ThreeLayerConvNet( num_filters = 3, filter_size = 3, input_dim = input_dim, hidden_dim = 7 )
loss, grads = model.loss( X, y )
for param_name in sorted( grads ) :
    f = lambda _: model.loss( X, y )[ 0 ]
    param_grad_num = eval_numerical_gradient( f, model.params[ param_name ], verbose = False, h = 1e-6 )
    e = rel_error( param_grad_num, grads[ param_name ] )
    print '%s  最大相对误差: %e' % ( param_name, rel_error(
                param_grad_num, grads[ param_name ] ) )
```

正确编码后的梯度检验结果：

```
W1  最大相对误差: 1.515827e-03
W2  最大相对误差: 6.044095e-03
W3  最大相对误差: 2.775748e-05
b1  最大相对误差: 4.377705e-05
b2  最大相对误差: 1.020082e-06
b3  最大相对误差: 1.250405e-09
```

- 过拟合少量数据

运行下列代码，确保在少量数据集上出现明显的过拟合现象，可视化示意图分别如图 6-13 和图 6-14 所示。

测试浅层卷积网络小数据过拟合代码块：

```
num_train = 100
small_data = {
  'X_train': data[ 'X_train' ][ : num_train ],
  'y_train': data[ 'y_train' ][ : num_train ],
  'X_val': data[ 'X_val' ],
  'y_val': data[ 'y_val' ],
}
model = ThreeLayerConvNet( weight_scale = 1e-2 )
```

```
trainer = Trainer( model, small_data,
                   num_epochs = 20, batch_size = 50,
                   update_rule = 'adam',
                   updater_config = { 'learning_rate': 1e-3, },
                   verbose = True, print_every = 5 )
trainer.train( )
```

训练结果：

```
(迭代 1 / 40) 损失值: 2.349292
(周期 0 / 20) 训练精度: 0.210000; 验证精度: 0.142000
(迭代 31 / 40) 损失值: 0.194024
(周期 16 / 20) 训练精度: 0.950000; 验证精度: 0.203000
(周期 17 / 20) 训练精度: 0.970000; 验证精度: 0.210000
(迭代 36 / 40) 损失值: 0.054560
(周期 18 / 20) 训练精度: 0.970000; 验证精度: 0.204000
(周期 19 / 20) 训练精度: 1.000000; 验证精度: 0.193000
(周期 20 / 20) 训练精度: 1.000000; 验证精度: 0.216000
```

可视化训练结果：

```
plt.subplot( 2, 1, 1 )
plt.title( 'Training loss', fontsize = 18 )
plt.plot( trainer.loss_history, 'o' )
plt.xlabel( 'iteration', fontsize = 18 )
plt.ylabel( 'loss', fontsize = 18 )
plt.subplot( 2, 1, 2 )
plt.subplots_adjust( left = 0.08, right = 0.95, wspace = 0.25, hspace = 0.3 )
plt.title( 'train accuracy VS val accuracy', fontsize = 18 )
plt.plot( trainer.train_acc_history, '-o' )
plt.plot( trainer.val_acc_history, '-*' )
plt.legend( [ 'train', 'val' ], loc = 'upper left' )
plt.xlabel( 'epoch', fontsize = 18 )
plt.ylabel( 'accuracy', fontsize = 18 )
plt.show( )
```

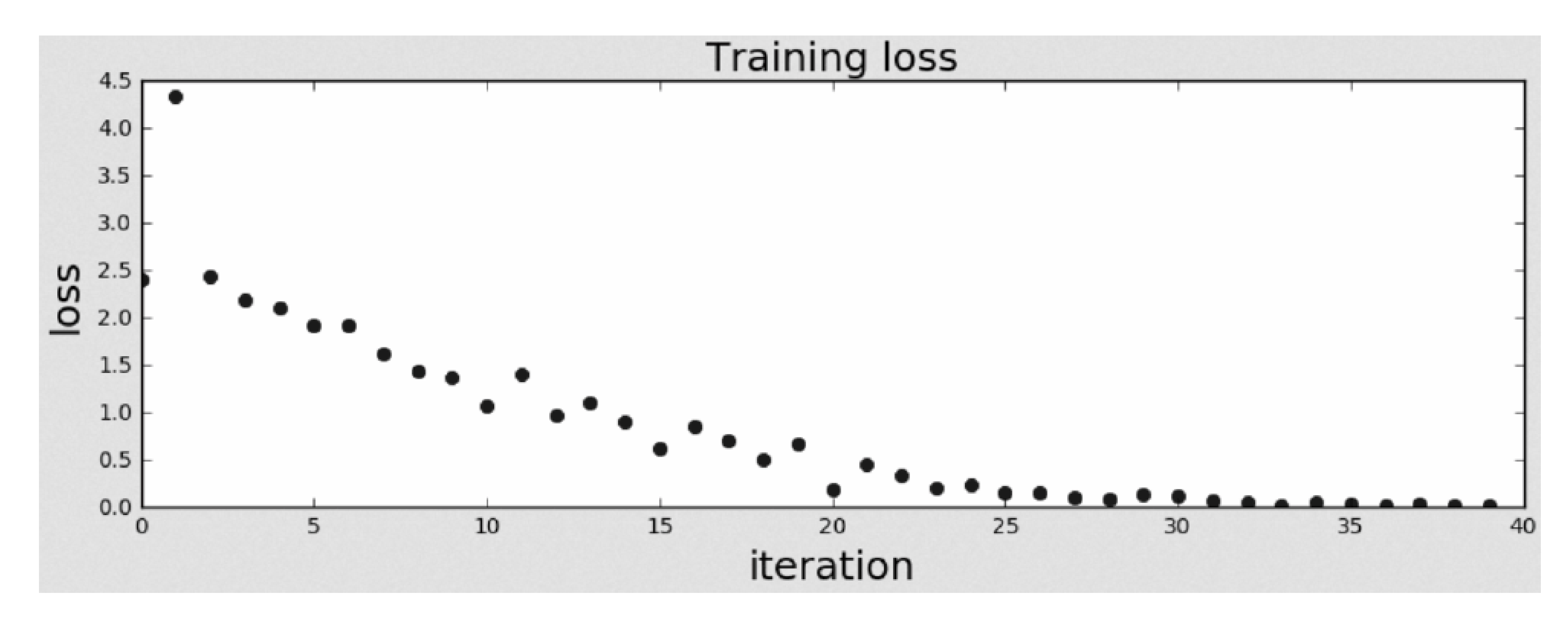


图 6-13　损失函数变化曲线

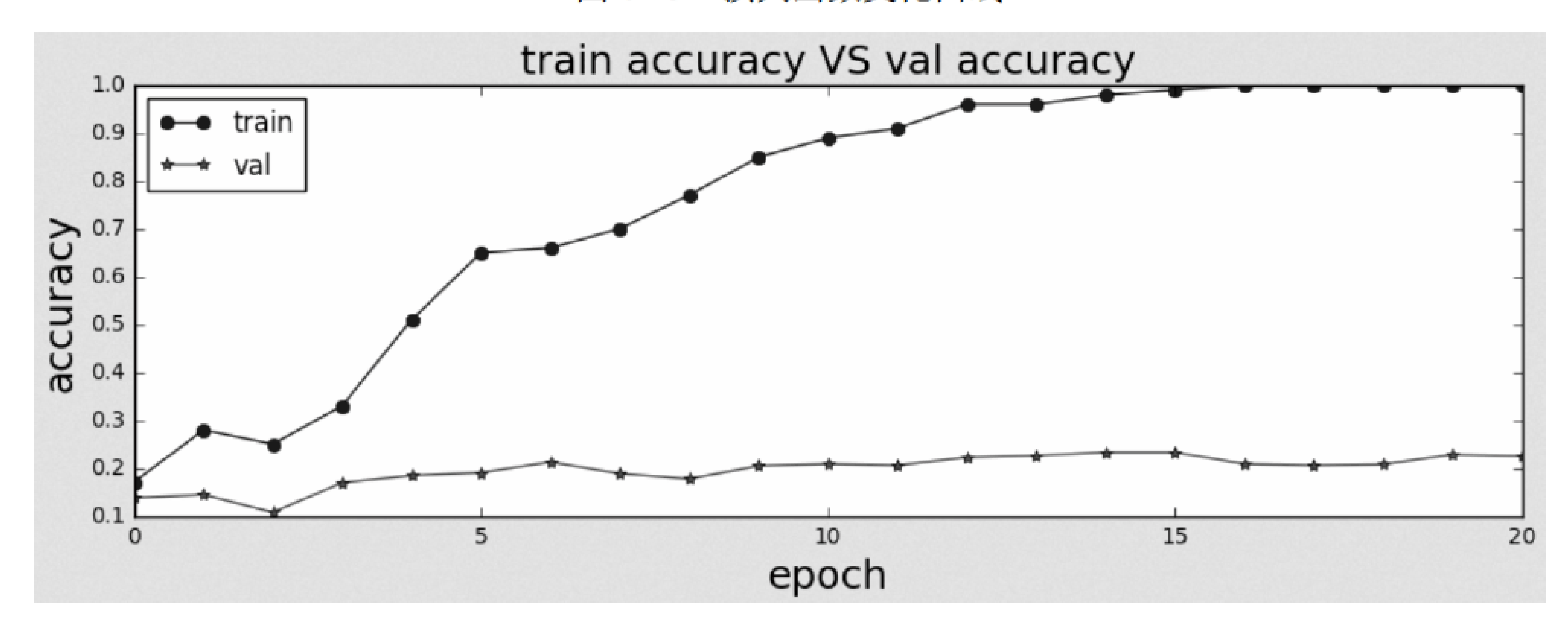


图 6-14　训练精度、验证精度变化曲线

6.5.8　空间批量归一化

BN[10]算法是一种高效实用的技术，大大加快了网络训练。但 BN 算法通常使用在全连接网络中，因此在卷积网络中会有一点修改，我们将其称为**空间批量归一化**（Spatial Batch Normalization，SBN）。默认情况下，BN 接收（N,D）数据进行批量归一化操作，因此，我们需要将 BN 算法调整为接收（N,C,H,W）数据。由于之前我们已经编写好了 BN 算法，现在只需要将图片数据重塑成（N,D），然后调用实现好了的 BN 算法，之后再将其输出结果重塑回（N,C,H,W）即可。

- SBN 前向传播

打开"DLAction/classifiers/chapter6/cnn_layers.py"文件，实现 spatial_batchnorm_forward 函数，完成后运行检验代码进行测试。

spatial_batchnorm_forward 函数代码块：

```
def spatial_batchnorm_forward( x, gamma, beta, bn_param ) :
    """
    空间批量归一化前向传播。
    Inputs:
```

```
- x: 数据 ( N, C, H, W )。
- gamma: 缩放因子 ( C, )。
- beta: 偏移因子 ( C, )。
- bn_param: 参数字典：
  - mode: 'train' or 'test';
  - eps: 数值稳定常数。
  - momentum: 运行平均值衰减因子。
  - running_mean: 形状为( D, ) 的运行均值。
  - running_var : 形状为 ( D, ) 的运行方差。
Returns 元组:
- out:输出 ( N, C, H, W )。
- cache: 用于反向传播的缓存。
"""
out, cache = None, None
#######################################################################
#                    任务：实现空间 BN 算法前向传播。                      #
#        提示：只需要重塑数据，调用 batchnorm_forward 函数即可。           #
#######################################################################




#######################################################################
#                                结束编码                              #
#######################################################################
return out, cache
```

完成上述代码后，使用下列代码进行测试，首先我们测试训练模式。

测试 SBN 前向传播训练模式代码块：

```
# 训练阶段：SBN 前向传播。
N, C, H, W = 2, 3, 4, 5
x = 4 * np.random.randn( N, C, H, W ) + 10
print '使用 BN 之前:'
print '    数据形状: ', x.shape
print '    均值: ', x.mean( axis = ( 0, 2, 3 ) )
print '    标准差: ', x.std( axis = ( 0, 2, 3 ) )
gamma, beta = np.ones( C ), np.zeros( C )
bn_param = { 'mode': 'train' }
out, _ = spatial_batchnorm_forward( x, gamma, beta, bn_param )
print '使用 BN 后:'
```

```
print '　输出数据形状: ', out.shape
print '　均值: ', out.mean( axis = ( 0, 2, 3 ) )
print '　标准差: ', out.std( axis = ( 0, 2, 3 ) )
gamma, beta = np.asarray( [ 3, 4, 5 ] ), np.asarray( [ 6, 7, 8 ] )
out, _ = spatial_batchnorm_forward( x, gamma, beta, bn_param )
print '在 BN 后使用(gamma, beta)进行缩放:'
print '　输出数据形状: ', out.shape
print '　均值: ', out.mean( axis = ( 0, 2, 3 ) )
print '　标准差: ', out.std( axis = ( 0, 2, 3 ) )
```

训练阶段 SBN 前向传播测试结果:

```
使用 BN 之前:
  数据形状:  ( 2L, 3L, 4L, 5L )
  均值:  [ 10.66668148   10.72390629   10.26857762 ]
  标准差:  [ 4.11024553   4.02371795   3.75411333 ]
使用 BN 后:
  输出数据形状:  ( 2L, 3L, 4L, 5L )
  均值:  [ -5.49560397e-16    1.85962357e-16   -7.91033905e-16 ]
  标准差:  [ 0.9999997    0.99999969   0.99999965 ]
在 BN 后使用( gamma, beta )进行缩放:
  输出数据形状:  ( 2L, 3L, 4L, 5L )
  均值:  [ 6.   7.   8. ]
  标准差:  [ 2.99999911   3.99999876   4.99999823 ]
```

接下来，我们测试 SBN 前向传播中的测试模式。

测试 SBN 前向传播测试模式代码块:

```
# 测试阶段：SBN 前向传播。
N, C, H, W = 10, 4, 11, 12
bn_param = { 'mode': 'train' }
gamma = np.ones( C )
beta = np.zeros( C )
for t in xrange( 50 ) :
  x = 2.3 * np.random.randn( N, C, H, W ) + 13
  spatial_batchnorm_forward( x, gamma, beta, bn_param )
bn_param[ 'mode' ] = 'test'
x = 2.3 * np.random.randn( N, C, H, W ) + 13
a_norm, _ = spatial_batchnorm_forward( x, gamma, beta, bn_param )
print '　均值: ', a_norm.mean( axis = ( 0, 2, 3 ) )
print '　标准差: ', a_norm.std( axis = ( 0, 2, 3 ) )
```

SBN 测试模式检验结果：
均值： [0.02892776 0.0520456 0.0546018 0.01706541]
标准差： [0.99260738 0.99075435 0.97366721 0.9950839]

- SBN 反向传播

接下来打开“DLAction/classifiers/chapter6/cnn_layers.py”文件，实现 spatial_batchnorm_backward 函数，完成后运行梯度检验代码进行测试。

spatial_batchnorm_backward 函数代码块：

```
def spatial_batchnorm_backward( dout, cache ) :
    """
     空间批量归一化反向传播。
    Inputs:
    - dout: 上层梯度( N, C, H, W )。
    - cache: 前向传播缓存。
    Returns 元组:
    - dx:输入梯度( N, C, H, W )。
    - dgamma: gamma 梯度( C, )。
    - dbeta: beta 梯度 ( C, )。
    """
    dx, dgamma, dbeta = None, None, None
    ################################################################
    #           任务：实现空间 BN 算法反向传播。                        #
    #       提示：只需要重塑数据调用 batchnorm_backward_alt 函数即可。  #
    ################################################################




    ################################################################
    #                           结束编码                            #
    ################################################################
    return dx, dgamma, dbeta
```

完成上述代码后，使用下列代码进行梯度检验。

SBN 反向传播梯度检验代码块：

```
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn( N, C, H, W ) + 12
gamma = np.random.randn( C )
beta = np.random.randn( C )
```

```
dout = np.random.randn( N, C, H, W )
bn_param = { 'mode': 'train' }
fx = lambda x: spatial_batchnorm_forward( x, gamma, beta, bn_param )[ 0 ]
fg = lambda a: spatial_batchnorm_forward( x, gamma, beta, bn_param )[ 0 ]
fb = lambda b: spatial_batchnorm_forward( x, gamma, beta, bn_param )[ 0 ]
dx_num = eval_numerical_gradient_array( fx, x, dout )
da_num = eval_numerical_gradient_array( fg, gamma, dout )
db_num = eval_numerical_gradient_array( fb, beta, dout )
_, cache = spatial_batchnorm_forward( x, gamma, beta, bn_param )
dx, dgamma, dbeta = spatial_batchnorm_backward( dout, cache )
print 'dx  误差: ', rel_error( dx_num, dx )
print 'dgamma  误差: ', rel_error( da_num, dgamma )
print 'dbeta  误差: ', rel_error( db_num, dbeta )
```

SBN 反向传播梯度检验结果：

```
dx  误差:    1.65364732287e-08
dgamma  误差:    2.37670881079e-12
dbeta  误差:    1.47269855681e-11
```

6.6　参考代码

卷积前向传播代码块：

```
def conv_forward_naive( x, w, b, conv_param ) :
    out = None
    N, C, H, W = x.shape
    F, _, HH, WW = w.shape
    stride, pad = conv_param[ 'stride' ], conv_param[ 'pad' ]
    H_out = 1 + ( H + 2 * pad – HH ) / stride
    W_out = 1 + ( W + 2 * pad – WW ) / stride
    out = np.zeros( ( N , F , H_out, W_out ) )
    x_pad = np.pad( x, ( ( 0, ), ( 0, ), ( pad, ) , ( pad , ) ),
mode = 'constant', constant_values = 0 )
    for i in range( H_out ) :
        for j in range( W_out ) :
            x_pad_masked = x_pad[ : , : , i * stride : i * stride + HH, j * stride : j * stride + WW ]
            for k in range( F ) :
                out[ : , k , i, j ] = np.sum(
                        x_pad_masked * w[ k, : , : , : ], axis = ( 1, 2, 3 ) )
    out = out + ( b )[ None, :, None, None ]
```

```
    cache = ( x, w, b, conv_param )
    return out, cache
```

conv_backward_naive 函数代码块：

```
def conv_backward_naive( dout, cache ) :
    dx, dw, db = None, None, None
    x, w, b, conv_param = cache
    N, C, H, W = x.shape
    F, _, HH, WW = w.shape
    stride, pad = conv_param[ 'stride' ], conv_param[ 'pad' ]
    H_out = 1 + ( H + 2 * pad – HH ) / stride
    W_out = 1 + ( W + 2 * pad – WW ) / stride
    x_pad = np.pad( x, ( ( 0, ), ( 0, ), ( pad, ), ( pad, ) ),
    mode = 'constant', constant_values = 0 )
    dx = np.zeros_like( x )
    dx_pad = np.zeros_like( x_pad )
    dw = np.zeros_like( w )
    db = np.zeros_like( b )
    db = np.sum( dout, axis = ( 0, 2, 3 ) )
    x_pad = np.pad( x, ( ( 0, ), ( 0, ), ( pad, ), ( pad, ) ),
    mode = 'constant', constant_values = 0 )
    for i in range( H_out ) :
        for j in range( W_out ) :
            x_pad_masked = x_pad[ : , : , i * stride : i * stride + HH, j * stride : j * stride + WW ]
            for k in range(F): #计算 dw
                dw[ k , : , : , : ] + = np.sum(x_pad_masked *
                                        ( dout[ : , k , i, j ] ) [ : ,None, None, None ], axis = 0 )
            for n in range( N ) : #计算 dx_pad
                dx_pad[ n , : , i * stride : i * stride + HH, j * stride : j * stride + WW ] + = np.sum(
                                    ( w[ : , : , : , : ] * ( dout[ n, : , i , j ] )[ : , None ,None, None ] ), axis = 0 )
    dx = dx_pad[ : , : , pad : -pad , pad : -pad ]
    return dx, dw, db
```

max_pool_forward_naive 函数代码块：

```
def max_pool_forward_naive( x, pool_param ) :
    out = None
    N, C, H, W = x.shape
    HH = pool_param[ 'pool_height' ]
    WW = pool_param[ 'pool_width' ]
```

```
    stride = pool_param[ 'stride' ]
    H_out = ( H – HH ) / stride + 1
    W_out = ( W – WW ) / stride + 1
    out = np.zeros( ( N, C, H_out, W_out ) )
    for i in xrange( H_out ) :
            for j in xrange( W_out ) :
                x_masked = x[ : , : , i * stride : i * stride + HH, j * stride : j * stride + WW ]
                out[ : , : , i , j ] = np.max( x_masked, axis = ( 2, 3 ) )
    cache = ( x, pool_param )
    return out, cache
```

max_pool_backward_naive 函数代码块：

```
def max_pool_backward_naive( dout, cache ) :
    dx = None
    x, pool_param = cache
    N, C, H, W = x.shape
    HH = pool_param[ 'pool_height' ]
    WW = pool_param[ 'pool_width' ]
    stride = pool_param[ 'stride' ]
    H_out = ( H – HH ) / stride + 1
    W_out = ( W – WW ) / stride + 1
    dx = np.zeros_like( x )
    for i in xrange( H_out ) :
        for j in xrange( W_out ) :
            x_masked = x[ : , : , i * stride : i * stride + HH, j * stride : j * stride + WW ]
            max_x_masked = np.max( x_masked, axis = ( 2, 3 ) )
            temp_binary_mask = ( x_masked == ( max_x_masked )[ : , : , None , None ] )
            dx[ : , : ,i * stride : i * stride + HH, j * stride : j * stride + WW ] + =
                                    temp_binary_mask * ( dout[ : , : , i , j ] )[ : , : , None , None ]
    return dx
```

权重初始化代码块：

```
    def __init__( self, input_dim = ( 3, 32, 32 ), num_filters = 32, filter_size = 7,
                hidden_dim = 100, num_classes = 10, weight_scale = 1e-3, reg = 0.0, ):
        self.params = { }
        self.reg = reg
        C, H, W = input_dim
        self.params[ 'W1' ] = weight_scale * np.random.randn( num_filters, C, filter_size, filter_size )
        self.params[ 'b1' ] = np.zeros( num_filters )
```

```
        self.params[ 'W2' ] = weight_scale * np.random.randn( ( H / 2 ) * ( W / 2 ) * num_filters, hidden_dim )
        self.params[ 'b2' ] = np.zeros( hidden_dim )
        self.params[ 'W3' ] = weight_scale * np.random.randn( hidden_dim, num_classes )
        self.params['b3'] = np.zeros( num_classes )
```

损失函数代码块：

```
def loss( self, X, y = None ):
    W1, b1 = self.params[ 'W1' ], self.params[ 'b1' ]
    W2, b2 = self.params[ 'W2' ], self.params[ 'b2' ]
    W3, b3 = self.params[ 'W3' ], self.params[ 'b3' ]
    filter_size = W1.shape[ 2 ]
    conv_param = { 'stride': 1, 'pad': ( filter_size – 1 ) / 2 }
    pool_param = { 'pool_height': 2, 'pool_width': 2, 'stride': 2 }
    scores = None
    conv_forward_out_1, cache_forward_1 = conv_relu_pool_forward( X, self.params[ 'W1' ],
                                                self.params[ 'b1' ], conv_param, pool_param )
    affine_forward_out_2, cache_forward_2 = affine_forward( conv_forward_out_1,
                                                self.params[ 'W2' ], self.params[ 'b2' ] )
    affine_relu_2, cache_relu_2 = relu_forward( affine_forward_out_2 )
    scores, cache_forward_3 = affine_forward(affine_relu_2, self.params[ 'W3' ], self.params[ 'b3' ] )
    if y is None:
        return scores
    loss, grads = 0, { }
    loss, dout = softmax_loss( scores, y )
    loss + = self.reg * 0.5 * ( np.sum( self.params[ 'W1' ] ** 2 )
                        + np.sum( self.params[ 'W2' ] ** 2 )
                        + np.sum( self.params[ 'W3' ] ** 2 ) )
    dX3, grads[ 'W3' ], grads[ 'b3' ] = affine_backward( dout, cache_forward_3 )
    dX2 = relu_backward( dX3, cache_relu_2 )
    dX2, grads[ 'W2' ], grads[ 'b2' ] = affine_backward( dX2, cache_forward_2 )
    dX1, grads[ 'W1' ], grads[ 'b1' ] = conv_relu_pool_backward( dX2, cache_forward_1 )
    grads[ 'W3' ] = grads[ 'W3' ] + self.reg * self.params[ 'W3' ]
    grads[ 'W2' ] = grads[ 'W2' ] + self.reg * self.params[ 'W2' ]
    grads[ 'W1' ] = grads[ 'W1' ] + self.reg * self.params[ 'W1' ]
    return loss, grads
```

spatial_batchnorm_forward 函数代码块：

```
def spatial_batchnorm_forward( x, gamma, beta, bn_param ) :
    out, cache = None, None
```

```
N, C, H, W = x.shape
temp_output, cache = batchnorm_forward(x.transpose( 0, 3, 2, 1 ).reshape( ( N * H * W, C ) ),
                                       gamma, beta, bn_param )
out = temp_output.reshape( N, W, H, C ).transpose( 0, 3, 2, 1 )
return out, cache
```

spatial_batchnorm_backward 函数代码块：

```
def spatial_batchnorm_backward( dout, cache ) :
    dx, dgamma, dbeta = None, None, None
    N,C,H,W = dout.shape
    dx_temp, dgamma, dbeta = batchnorm_backward_alt(
          dout.transpose( 0, 3, 2, 1 ).reshape( ( N * H * W, C ) ), cache )
    dx = dx_temp.reshape( N, W, H, C ).transpose( 0, 3, 2, 1 )
    return dx, dgamma, dbeta
```

6.7　参考文献

[1] Lecun, Y., & Bengio, Y. (1998). Convolutional networks for images, speech, and time series: MIT Press.

[2] Hubel, D. H., & Wiesel, T. N. (1959). Receptive fields of single neurones in the cat's striate cortex. Journal of Physiology, 148(3), 574.

[3] Goodfellow, I. J., Wardefarley, D., Mirza, M., Courville, A., & Bengio, Y. (2013). Maxout Networks. Computer Science, 1319-1327.

[4] Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. Neural Networks, 2(5), 359-366.

[5] Thom, M., & Palm, G. (2016). Sparse activity and sparse connectivity in supervised learning. Journal of Machine Learning Research, 14(1), 1091-1143.

[6] Zhou, Y. T., & Chellappa, R. (1988). Computation of optical flow using a neural network. Paper presented at the IEEE International Conference on Neural Networks.

[7] Zubair, S., Yan, F., & Wang, W. (2013). Dictionary learning based sparse coefficients for audio classification with max and average pooling. Digital Signal Processing, 23(3), 960–970.

[8] Lecun, Y. (1989). Generalization and Network Design Strategies. Paper presented at the Connectionism in Perspective.

[9] Gregor, K., & Lecun, Y. (2010). Emergence of Complex-Like Cells in a Temporal Product Network with Local Receptive Fields. Computer Science.

[10] Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Computer Science.

第 7 章

循环神经网络

在本书前面章节的介绍中，神经网络总是单向的，从输入层到低级隐藏层，再从低级隐藏层到高级隐藏层，最后再到输出层。但不管网络有多少层，都是一层一层地前向输出。但这其实是有问题的，因为这种前馈结构需要**假设数据是独立同分布**，但现实中有很多复杂的数据都不满足这个条件，例如音频数据、视频数据及自然语言数据等。当我们将一篇英文文本翻译成中文文本时，句子之间，段落之间是存在上下文关系的，在特定的情景中单词的意义也不一样。并且这些数据的长度还不一样，我们很难规格化这些序列数据去训练，因此前馈网络很难处理这类变长的序列数据。当然，我们也可以将整篇文章当作一条文本数据，将整段音频当作一条音频数据一次性输入前馈网络中学习，但这样的做法数据维度实在高得难以想象，并会产生可怕的**维度灾难**（Curse of Dimensionality）[1]。另一种方式就是对序列数据进行切割，例如将机器翻译任务中的一篇文章切成段落训练，或者将段落切成句子训练，将手写识别的句子切成一个个的单词进行识别训练。但这需要很强的人为干预，数据切分越细致，数据的上下文关系就破坏得越严重，并且数据切分不合理也会引入很强的人为噪声。

为了有效地处理这类序列数据，我们允许神经网络进行**横向连接**。当前的网络输出不仅依赖于当前的输入信息，还依赖之前的数据信息，而这类网络结构就是本章的主角——**循环神经网络**（Recurrent Neural Network，RNN）[2]。

在 7.1 节中，我们将把循环神经网络展开，仔细介绍循环网络的前向传播以及反向传播过程。

在 7.2 节中，我们将会介绍循环网络的一些变种，典型的结构有双向循环网络、编码-解

码网络和深度循环网络等。

在 7.3 节中，我们将重点介绍一种带有门控功能的特殊循环神经网络——LSTM，它是实际应用中效果最佳的循环网络之一。

在 7.4 节中，我们将一步一步地完成简单 RNN 网络的前向传播与反向传播编码，然后使用 RNN 完成自动图片说明任务。

在 7.5 节中，我们将完成 LSTM 网络的前向传播与反向传播编码，然后使用其完成自动图片说明任务。

这里需要注意的是，循环神经网络有时也会被翻译成递归神经网络，但这对于真正的**递归网络**（Recursive Neural Network）[3]是不公平的。循环网络代表信息在时间维度从前往后的传递和积累，其展开是一个**链式结构**，应用在机器翻译中是假设句子后面的信息和句子之前的信息有关。而递归网络在空间维度的展开是一个树结构，应用在机器翻译中就是假设句子是一个树状结构，由几个部分（主语，谓语，宾语）组成，每个部分又可以再分成几个小部分，即某一部分的信息由它的子树组合而来。本章我们介绍的是第一种**循环网络**（Recurrent Neural Network），并不涉及**递归网络**（Recursive Neural Network）。

7.1　循环神经网络

循环神经网络的结构其实非常简单，相较典型的前馈神经网络，仅仅将网络隐藏层的输出（也可以是输出层的输出）重新连接回隐藏层，形成一个闭环。我们也可以理解成是在前馈网络中加入了**记忆单元：**当神经元前向传播时，隐藏层除了向网络的前端输出信息，还会将信息保存在记忆单元中。当执行下一条数据时，会将下一条数据与当前存储在记忆单元的信息，一起输入到隐藏层中进行特征提取，然后再将处理后的信息保存进记忆单元。

假设有一条长度为 T 的序列 X：I 和 H 分别表示 RNN 的输入单元与隐藏单元数量，如式（7.1）和（7.2）所示，为循环网络的前馈输出。

$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1} \tag{7.1}$$

$$b_h^t = f(a_h^t) \tag{7.2}$$

其中 x_i^t 表示第 t 时刻序列数据 x 的第 i 维；a_j^t 和 b_j^t 分别表示 t 时刻第 j 隐藏单元的输入值与激活值；w_{ih} 表示输入层第 i 单元与隐藏层 h 单元的连接权重；$w_{h'h}$ 表示隐藏层 h' 单元与隐藏层 h 单元的连接权重；$b_{h'}^{t-1}$ 表示 t-1 时刻隐藏层 h' 单元的激活值。

需要注意的是，在第 t=1 时刻，也就是序列第一条数据输入网络时，隐藏层还没有激活值，因此我们会设置 $b_{h'}^0$=0。当然，如果我们执行类似图像说明这样的联合 RNN 与 CNN 的任务时，$b_{h'}^0$ 就为卷积网络的输出特征。

7.1.1　循环神经元展开

为了更好地说明循环神经网络的工作模式，如图 7-1 所示，我们将序列数据按照时间维

度展开。假设我们的序列长度为 t，那么该神经网络就有 t 层隐藏层，t 层输入层。其中隐藏层之间的连接，以及输入层到隐藏层的连接都是共享的。

如果以静态的方式看待循环神经网络，那该网络就非常的“怪异”：虽然它有 t 层隐藏层，是非常深的神经网络，但却层层参数共享，实际上只有三套参数，**输入层到隐藏层参数 U，隐藏层到隐藏层参数 W，隐藏层到输出层参数 V**。

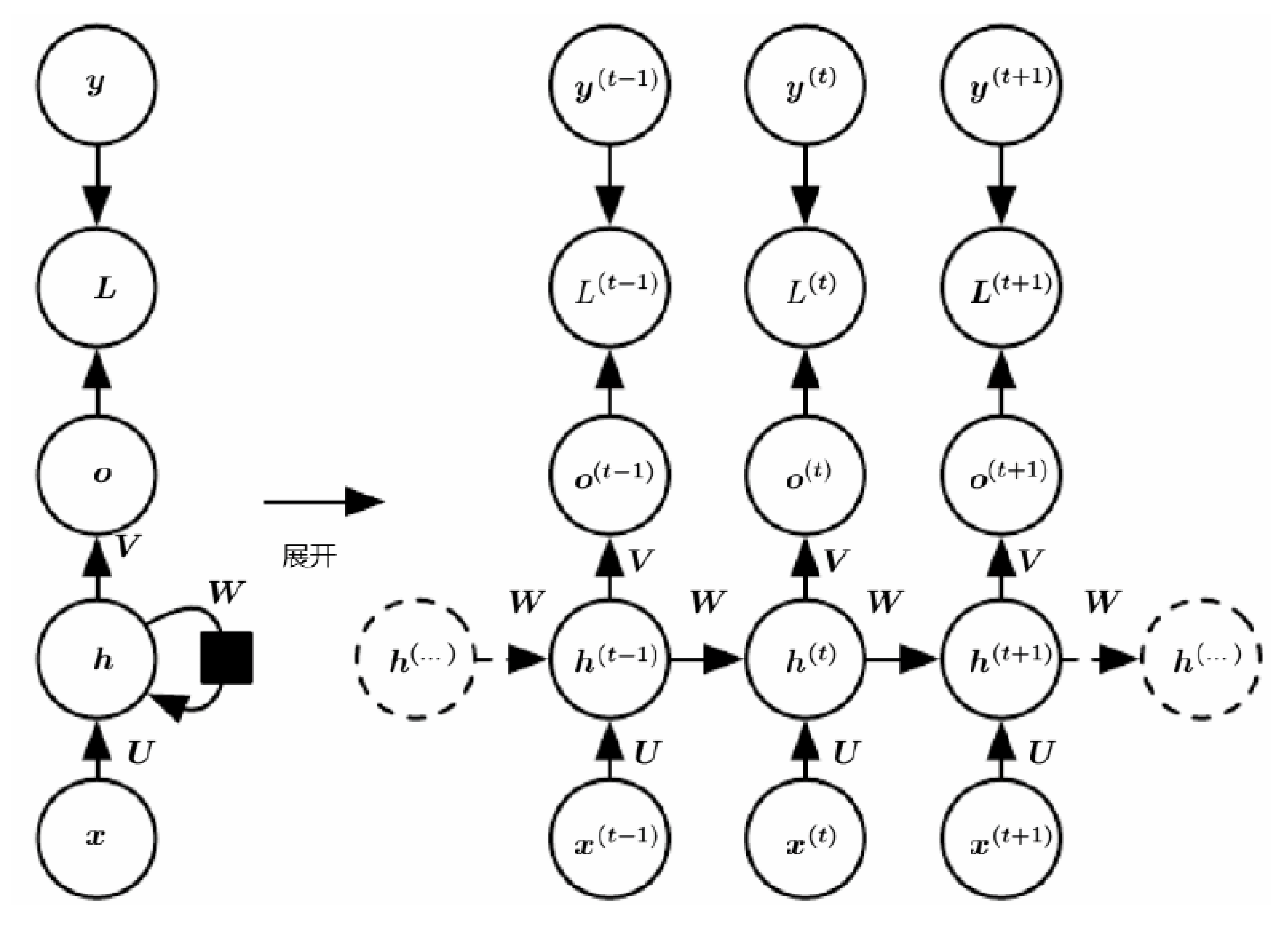


图 7-1　循环神经网络展开示意图

这样的参数共享结构有以下两个主要优点。

1. 无论序列的长度多长，学习模型始终具有相同大小的输入。
2. 由于每个时间步的参数都相同，因此我们可以使用相同的转移函数学习。

这两个因素使得我们不需要针对每个时间片段配置特定的参数，学习单独的函数。我们只需要学习一个转移函数即可，这样既可以节约大量的参数，并且也有利于提升模型的泛化能力，可以很方便地使用网络处理任意长的序列数据。

7.1.2　循环网络训练

通常，我们使用**时间反向传播**（BackPropagation Through Time，BPTT）[4]算法训练循环神经网络。它是 BP 算法应用在循环网络上的一种扩展形式，如果将循环网络展开成一个深层参数共享网络，那么将此算法看作 BP 算法即可。如果对 BP 算法的整个过程还显生疏，建议再返回本书第 3 章，仔细地推导 20 世纪 80 年代起就统治着神经网络的伟大算法。

根据学习输出种类的不同，我们主要将 RNN 分为三种：固定长度到可变长度，可变长度到固定长度，可变长度到可变长度的学习。接下来我们就简单地介绍这三种训练模式。

- 固定长度到可变长度学习

如图 7-2 所示，我们将固定（特征）大小的数据传入到 RNN 中，然后将其输出为可变长

度的序列数据。比如本章我们将要完成的自动图片说明任务[5]就采用这种网络架构。在该任务中，我们首先会将图片经过卷积网络进行图片特征提取，然后将图片特征当作 RNN 的隐藏层初始状态进行学习，最后 RNN 会输出一段该图片的文字描述。

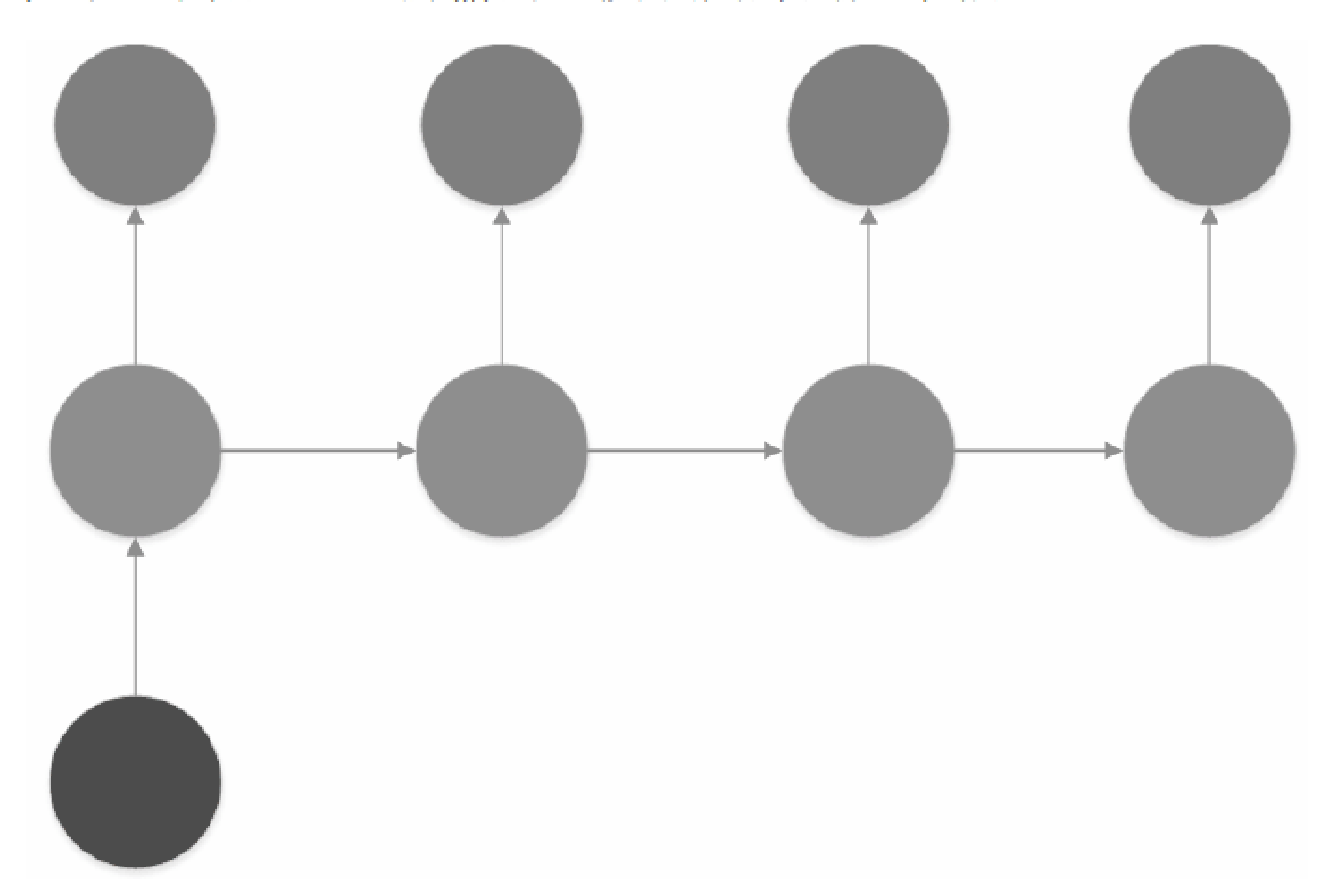


图 7-2　固定长度到可变长度学习示意图

- 可变长度到固定长度的学习

如图 7-3 所示，我们将序列数据映射到一个固定大小的隐藏层，在隐藏层中进行循环处理后，将最后隐藏层的状态输出到输出层。这种方式使得我们的序列可以由变长转化为固长，例如情感分类任务便是典型地将一段文字进行分类识别的任务。

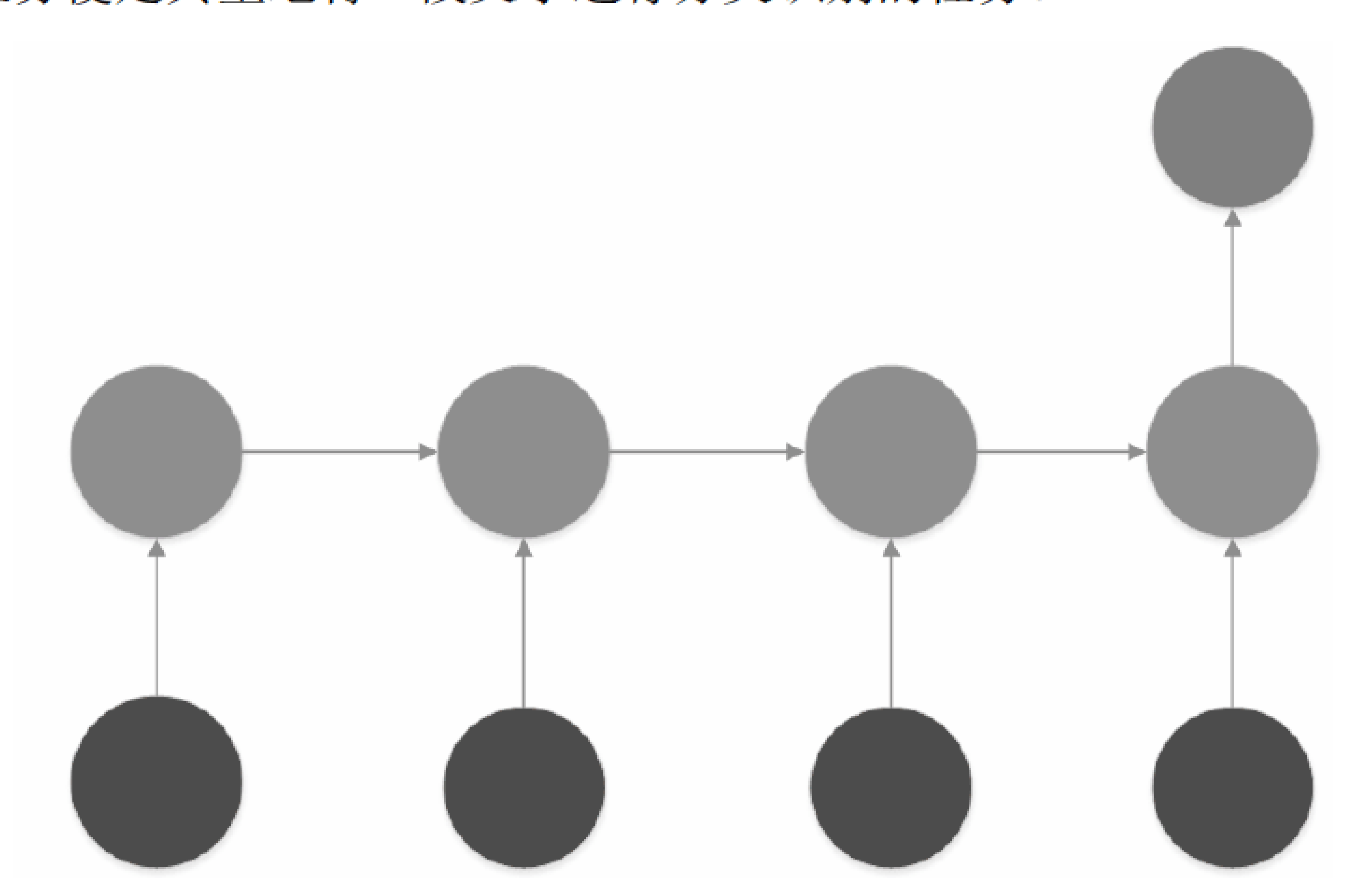


图 7-3　序列到固定长度表示学习示意图

这种序列到固定长度学习的任务训练起来也非常的方便，只需要将时间 t 看作是原始前馈网络的 t 层，然后使用 BP 算法训练这种按层共享参数的“深层神经网络”即可。

- 可变长度到可变长度的学习

如图 7-4 所示，序列的每一个时间片段都对应着一个输出结果，此时我们的任务就变成了学习如何映射输入序列到输出序列。

这类任务的训练方式和前面方法类似，都是将整个序列完全地输入到网络中，然后从序列的最后一个时间片段开始时间反向传播训练，只是在训练每个时间片段时，残差除了来自于上一时间片段，还来自于当前时间片段。

$$\delta_h^t = f'(a_h^t)(\sum_{k=1}^{K}\delta_k^t w_{hk} + \sum_{h'=1}^{K}\delta_{h'}^{t+1} w_{hh'}) \tag{7.3}$$

如式（7.3）所示，δ_h^t 表示第 t 时间片段，隐藏层 h 的残差。而该残差来自于当前时间片段 t 的各输出层残差与隐藏层到输出层连接权重乘积的累加，再加上 t+1 时间片段的各隐藏层残差与隐藏层到隐藏层连接权重乘积的累加。需要说明的是在 $t = T$，也就是训练序列最后一时间片段时，我们没有 $t = T + 1$ 时刻的残差，一般情况下我们会将其设置为 0，因此在 $t = T$ 时，我们仅需要计算当前时刻的输出层残差即可。

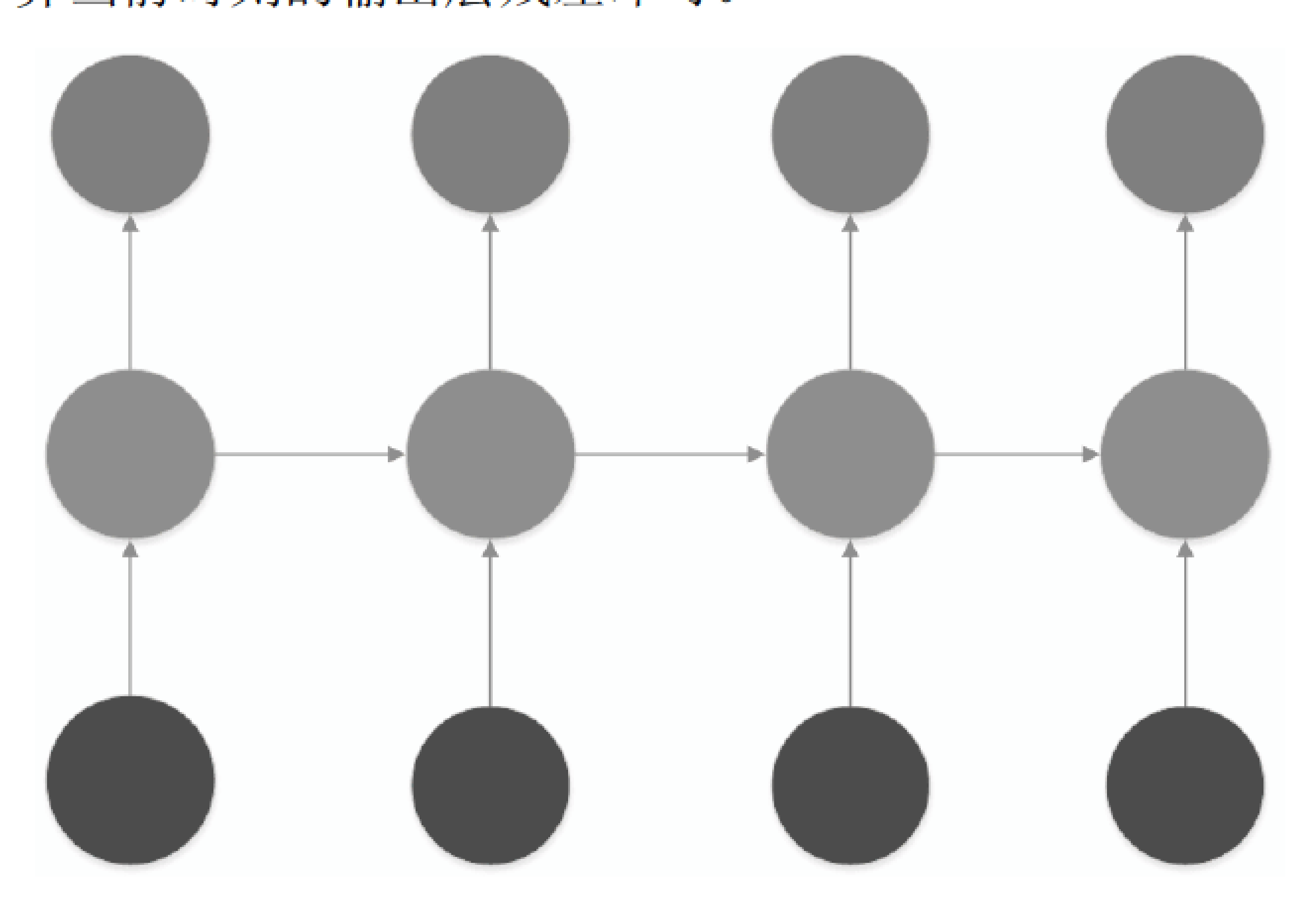


图 7-4　可变长度到可变长度学习示意图

7.2　循环神经网络设计

接下来，我们将会介绍循环网络的一些变种，典型的结构有双向循环网络，编码-解码网络及深度循环网络等。

7.2.1　双向循环网络结构

到目前为止，我们介绍的 RNN 总是将“过去”的信息整合起来，然后辅助处理当前信息。但在某些任务中，我们可能也需要整合“未来”的信息来处理当前信息。如在语音识别中，由于**协同发音**（co-articulation）的问题，当前的语音翻译可能需要依赖于接下来的一些**音素**（phonemes，最小的语音单位）。甚至由于一些语言的语法依赖关系，当前的音素也可能依赖于接下来的一些单词。在这类任务中，我们需要结合之前和之后的信息去消除歧义。**双向循环神经网络**（Bidirectional Recurrent Neural Networks，BRNNs）[6]是一种结合过去和未来信息处理当前信息的网络结构，目前该网络结构广泛地应用于手写识别[7]及语音识别领域[8]。

顾名思义，双向 RNN 其实就是由两个 RNN 同时组成，其中一个 RNN 前向处理序列数

据从序列的起始片段处理；另一个 RNN 反向处理序列数据，从序列的末尾片段开始，然后倒序处理。如图 7-5 所示，$h^{(t)}$ 表示顺序处理的子 RNN 中 t 时刻的隐藏单元，$g^{(t)}$ 表示逆序处理的子 RNN 中 t 时刻的隐藏单元，在计算 t 时刻的输出单元 $o^{(t)}$ 时，网络既考虑了之前的信息 $h^{(t)}$，又考虑了之后的信息 $g^{(t)}$。

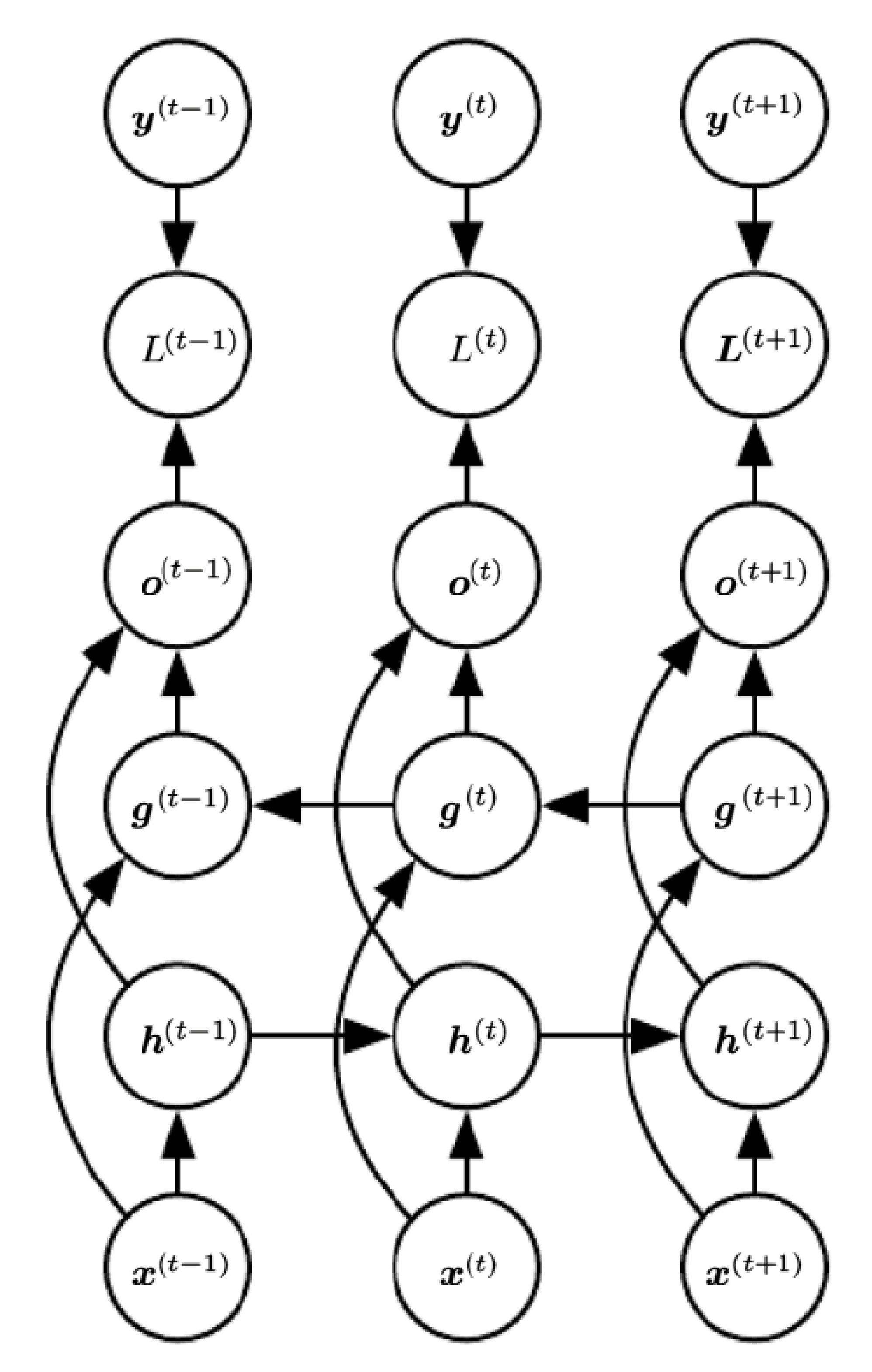


图 7-5　双向 RNN 结构示意图

7.2.2　编码-解码网络结构

在上一小节的可变长度到可变长度学习中，网络的输入和输出长度总是相同的，但就某些任务而言，这就显得非常不合理了。例如，在进行机器翻译任务时，我们不可能将英文序列整齐地翻译成中文序列，并且大多数情况下，英文序列的长度和中文序列的长度也应该不同。假设你临时被分配到给外国友人做翻译小哥，如何才能体现出你是一个英语大牛呢？外国友人说一个字，你翻译一个字，这种策略肯定不对（这就相当于等长的输入-输出模型）。最好的方式是他说完一句话，然后你在脑中出现这句话对应的一些概念，比如时间、地点、人物、事情、心情等，然后你再根据这些概念组成一句话翻译出来。这里的翻译更多的是一个再创造的过程，你获得了输入所要传达的一些抽象概念，然后根据自己的“喜好”转述成另一种语言，可以是直白的话，可以是一首诗，当然也可以是一串 Python 代码。

编码-解码（Encoder-Decoder）[9]或**序列到序列**（Sequence-to-Sequence）[10]结构便是最简单的一种映射**变长序列**（Variable-length Sequence）到变长序列的网络结构。需要说明的是，我们经常将 RNN 的输入称之为“**上下文**（context）”，而我们需要做的是生成该上下文 C

的某种表示。这种表示你可以理解成是由输入转化的一些特定概念，它可以是一个固定向量，也可以是一个序列向量。

你可能有些迷糊了，但静下心来就会发现这种结构其实非常的简单，如图 7-6 所示，编码-解码主要分为以下两个部分。

1. 我们将输入序列输入到一个称为**编码器**（Encoder）或**读者**（Reader）的 RNN 中，编码器将序列映射到一个向量 C 中，通常此向量为 RNN 隐藏层的最后状态。

2. 我们将向量 C 作为输入数据，输入到一个称为**解码器**（Decoder）或者**写者**（Writer）的 RNN 中，从而生成一条输出序列。

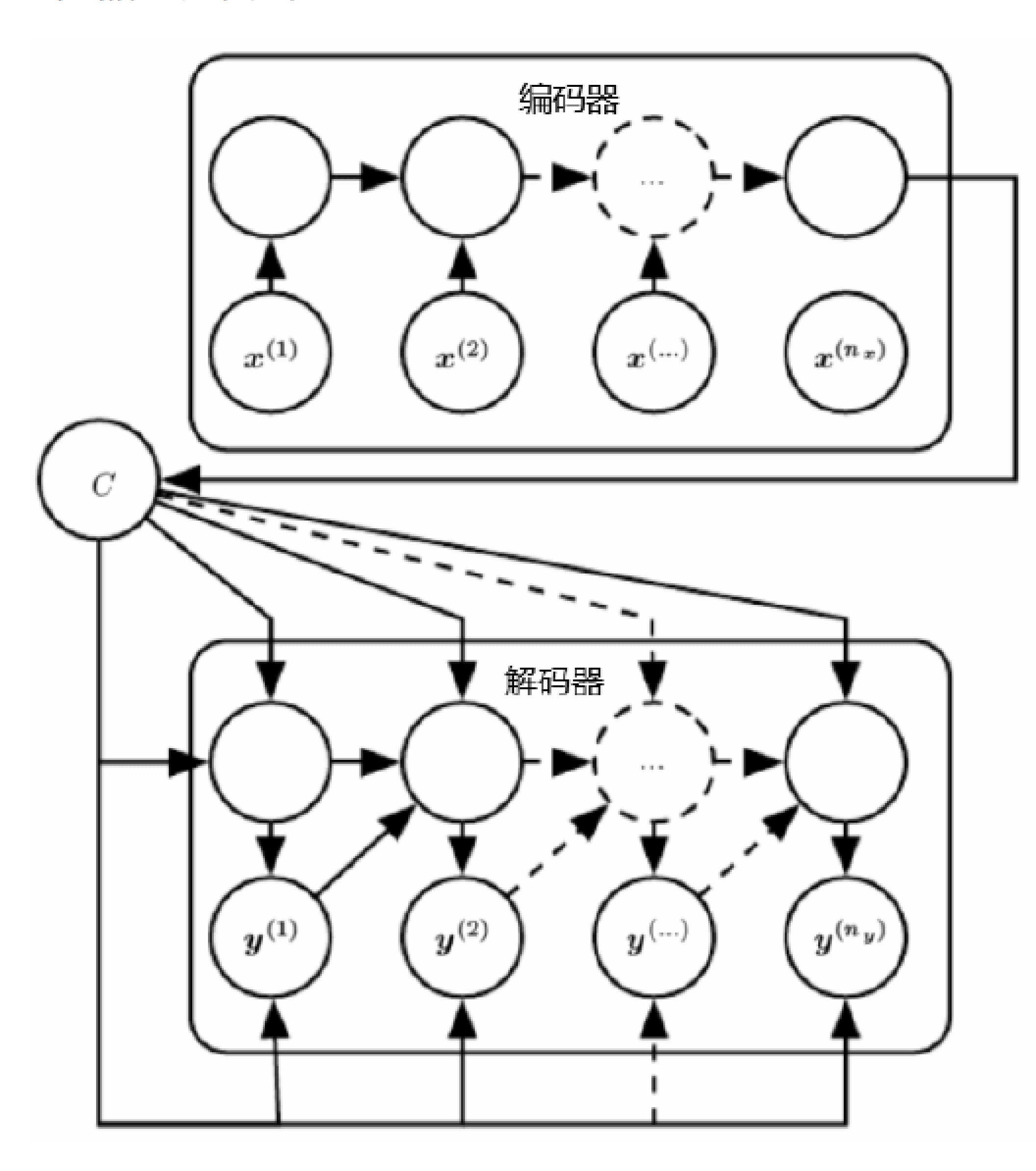


图 7-6　编码-解码 RNN 结构示意图

7.2.3　深度循环网络结构

RNN 一般可以分解成三个部分：一是输入层到隐藏层；二是前一隐藏层到下一隐藏层；三是隐藏层到输出层。这三个部分，我们可以用三组权重矩阵来表示。虽然从训练角度出发，将隐藏层单元按照时间展开可以看作是一个非常深的权重共享网络；但从模型能力的角度，其依然只能算浅层的网络。这就好像没有公主的命（不是深层结构），却得了公主的病（训练困难，梯度消失、爆炸等问题）。

在本书之前的章节中，我们已经强调多次深度结构带来的强大能力，那在 RNN 中加入深层的结构[11]会不会更好呢？答案是肯定的，但如何添加就是一个问题，本身 RNN 已经很难训练了，再加上深层的结构，那训练难度也就成倍地增长了。这里我们主要介绍三种常用的深层 RNN 结构作为参考。

如图 7-7（a）所示，我们将 RNN 中隐藏层扩展为多层 RNN 的结构[12]，h 和 z 分别表示不同的 RNN 隐藏层，h 层为低层循环网络，z 层为高层循环网络，其各自隐藏层的状态在各自的层中循环。

如图 7-7（b）所示，我们将 RNN 的三个部分分别看作是三个多层感知机[13]：首先，输入层到循环隐藏层会经过多层前馈网络；然后，循环隐藏层也是一个多层感知机，只有在循环隐藏层的输出才可以连接回循环隐藏层的输入。这就如同每一时间片段的序列都经过多级处理，然后再连接到前一层；最后，从隐藏层到输出层也是一个多层感知机，隐藏层的输出经过多级的前馈处理后再进行结果输出。

图 7-7（b）这种网络结构虽然大大增强了网络能力，但也变得非常难训练。如图 7-7（c）所示，为了缓解训练困难，我们也可以将循环隐藏层设计成越层循环[13]的方式。例如 h 层的一些神经元处理当前时间片段后，就会将结果存储起来作为下一时间片段使用，而一些神经元会经过多层的处理后再将当前时间片段存储起来。而下一时间片段，会综合单循环以及多级循环的信息。

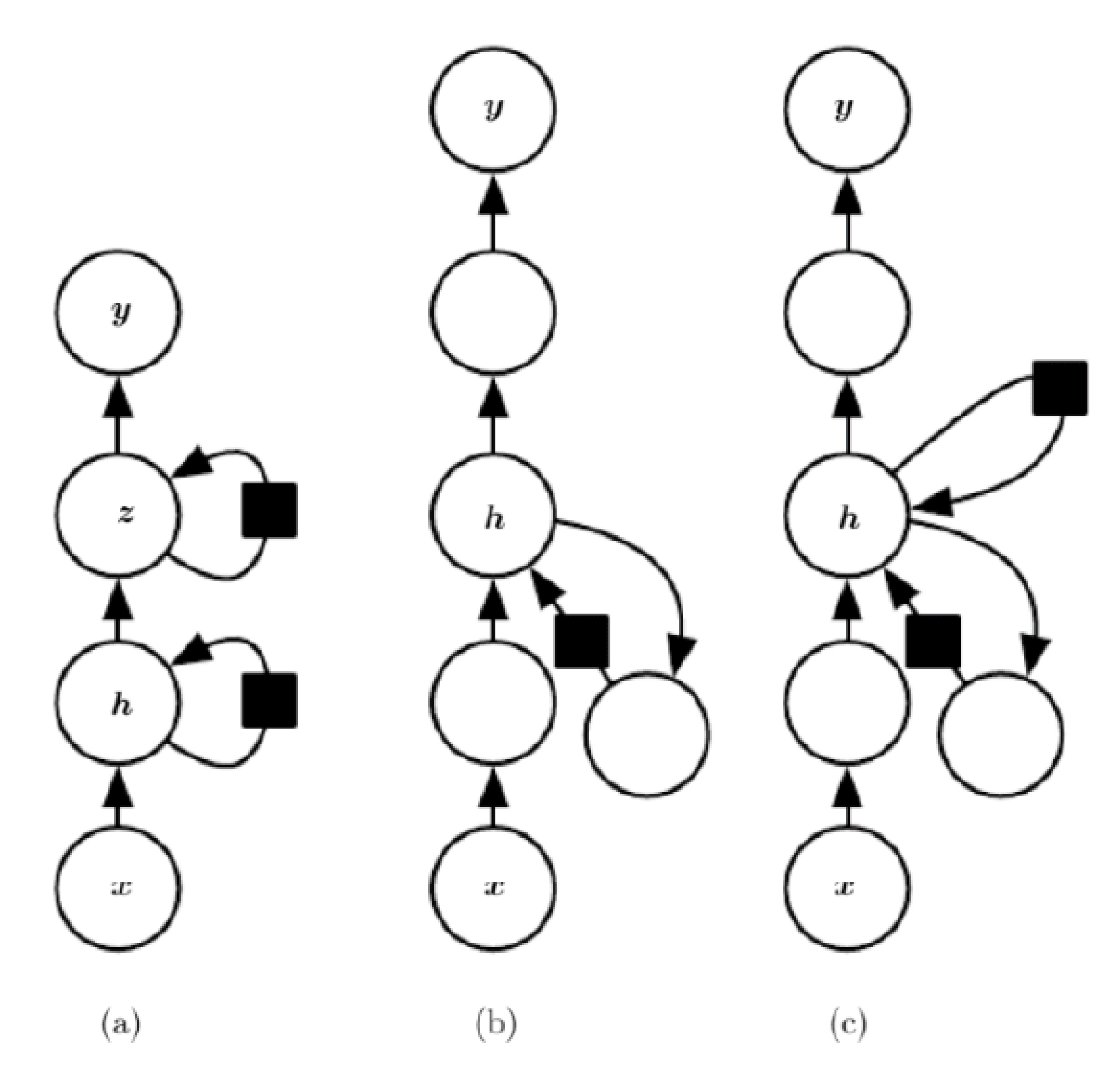


图 7-7　深层循环网络结构示意图

7.3　门控循环神经网络

循环网络有一个非常严重的缺陷，那就是很难学习**长期依赖**（Long-Term Dependencies）[14]。如图 7-8 所示，我们使用黑色圆圈表示关键信息，随着 RNN 的执行，当前的重要信息会不断地被“稀释”。当执行到第 7 时间片段时，最开始的信息几乎就不起作用了，这就是 RNN 的长期依赖困难。我们知道文章有上下文联系，但这并不意味着每段话，每个句子都是重要的，有时一篇文章需要抓住几句关键句子，几个关键词，便可以理解八成以上，反之则很难理解文本。但这些关键信息可能出现在序列数据的任何位置，可能是开头，可能是中段，也可能是末尾，如果关键信息之间的跨度很大，就会出现长期依赖问题。

在理想情况下，关键信息的位置不应该有区别，但 RNN 却像一条“快乐的小金鱼”，记忆力很差。当处理文本分类问题时，如果关键信息出现在末尾，那我们很容易就可以学习到这一信息。但如果这一信息出现在开头，学习起来就非常的困难。该问题的根本原因是一个老生常谈的问题——梯度消失或爆炸[15]，由于长时间（展开为深层）的跨度，梯度不断地消

耗，到达最前面时，几乎趋向为零，也就无法学习了。那如何才能缓解“记忆力”差呢？

图 7-8　RNN 长期依赖困难示意图

如果“记忆力”是一间小房子，那想要记住重要信息的最简单方法其实只是给记忆之门上把“锁”而已。“选择重要的，清除不重要的，锁住重要的”这就是本小节的重点——**长短期记忆网络**（Long Short-Term Memory，LSTM）[16]及其改进的基于**门控**（Gated）循环网络的核心思想。该类网络是目前实际应用中**最高效的序列模型**，并且已广泛应用于**语音识别**[17]、**手写识别**[7]、**机器翻译**[10]、**图像说明**[18]和**语法解析**[19]。接下来我们就详细地介绍该类网络。

7.3.1　LSTM

LSTM 由一组称之为**记忆块**（memory blocks）的循环子网构成，每个记忆块包含一个或多个自连接的记忆细胞及三个乘法控制单元——**输入门**、**输出门**和**遗忘门**[20]组成，提供着类似于**读**、**写**、**重置**的功能。如图 7-9 所示，是单个细胞的 LSTM 结构示意图。和 RNN 相似，其隐藏单元也是横向地连接回隐藏单元，只是将 RNN 的隐藏单元替换成了具有门控功能的记忆细胞。

输入门、输出门、遗忘门可以看作是三个 sigmoid 神经元，其输出为“0”或“1”，表示某项功能的“关闭”或“开启”。和 RNN 类似，其输入为数据的当前时间片段以及前一时间片段隐藏层的输出。当然，我们也可以加入如图 7-9 中的虚线所示，称之为**窥视孔**或**猫眼**（peephole）[21]连接，其表示门控单元不仅需要考虑当前的输入以及之前的输出，还需考虑自身存储的信息。

- **输入门**控制着当前信息的输入：当信息经过输入单元激活后会和输入门进行相乘，以确定是否写入当前信息；
- **遗忘门**控制着是否重置之前的记忆信息：其与细胞之前的记忆信息进行乘法运算，以确定是否保留之前的信息；
- **输出门**控制着当前记忆信息的输出：其与当前细胞记忆信息进行相乘以确定是否输出信息。

- **细胞单元**总是将当前的输入信息与之前的记忆信息进行**累加**。

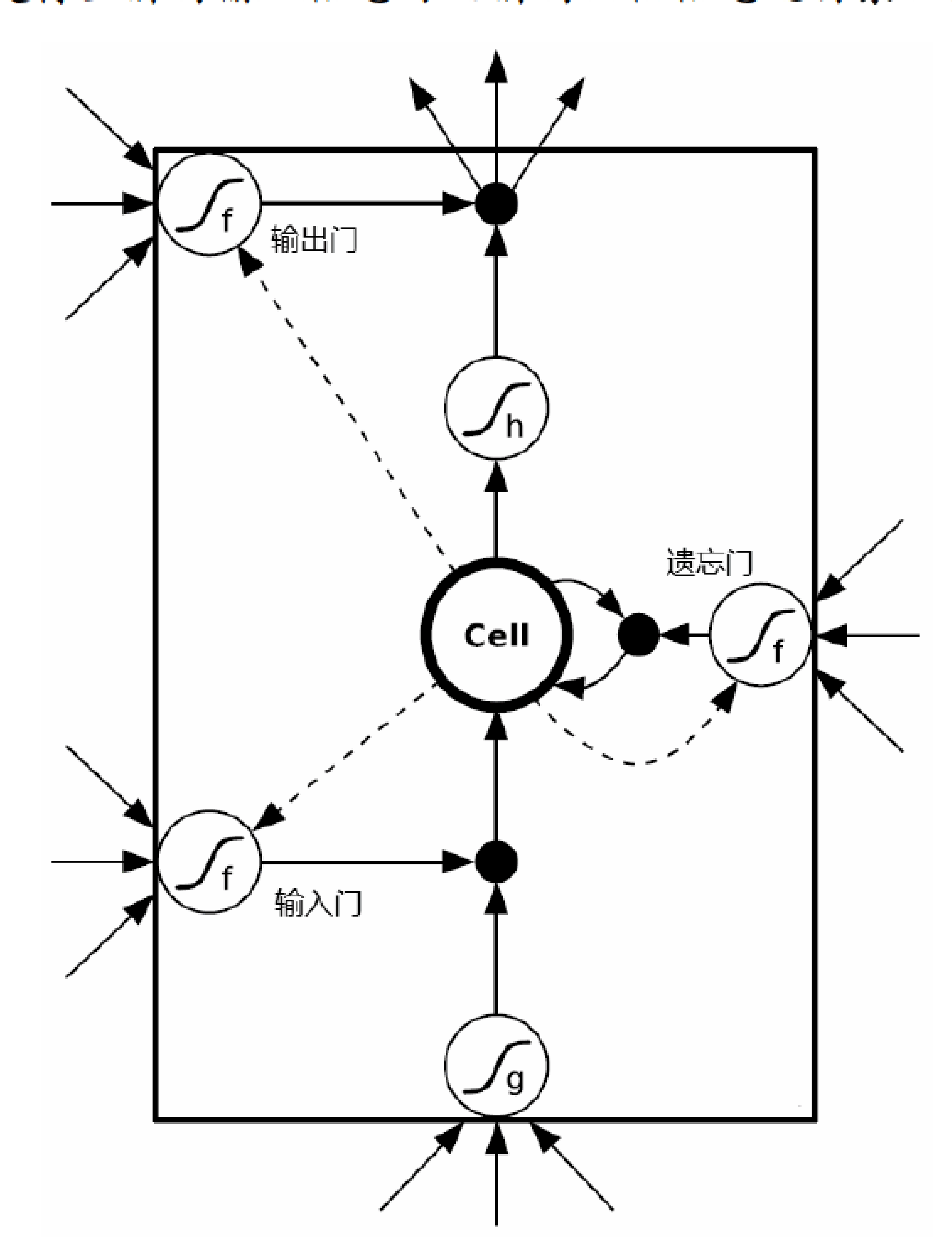


图 7-9　LSTM 结构示意图

我们将上述的文字稍微整理一下，如下所示，我们将带有 peephole 连接的 LSTM 网络的执行过程，以公式的形式再详细描述一遍，希望帮你理清思路。由于该结构字母公式较多，请在清醒时学习。

带 peephole 连接的 LSTM 的执行过程。
字符说明：a^t_{InGate}，a^t_{FGate}，a^t_{OutGate} 分别表示 t 时刻输入门，遗忘门，输出门的累计值；$f(x)$，$g(x)$，$h(x)$ 为激活函数；s 表示细胞的记忆信息；c 表示 LSTM 中的记忆细胞数量；I 表示输入层数量，H 表示隐藏层数量。 1.计算输入门： $$a^t_{\text{InGate}}=\sum_{i=1}^{I}w_{i\text{InGate}}x^t_i+\sum_{h=1}^{H}w_{h\text{InGate}}b^{t\text{-}1}_h+\sum_{c=1}^{C}w_{c\text{InGate}}s^{t\text{-}1}_c$$ $$b^t_{\text{InGate}}=f(a^t_{\text{InGate}})$$ 2.计算输入单元： $$b^t_{\text{Input}}=g(\sum_{i=1}^{I}w_{i\text{Input}}x^t_i+\sum_{h=1}^{H}w_{h\text{Input}}b^{t\text{-}1}_h)$$ 3.计算遗忘门：

$$a_{\text{FGate}}^t = \sum_{i=1}^{I} w_{i\text{FGate}} x_i^t + \sum_{h=1}^{H} w_{h\text{FGate}} b_h^{t-1} + \sum_{c=1}^{C} w_{c\text{FGate}} s_c^{t-1}$$

$$b_{\text{FGate}}^t = f(a_{\text{FGate}}^t)$$

4.更新细胞记忆：

$$s_c^t = b_{\text{FGate}}^t s_c^{t-1} + b_{\text{InGate}}^t b_{\text{Input}}^t$$

5.计算输出门：

$$a_{\text{OutGate}}^t = \sum_{i=1}^{I} w_{i\text{OutGate}} x_i^t + \sum_{h=1}^{H} w_{h\text{OutGate}} b_h^{t-1} + \sum_{c=1}^{C} w_{c\text{OutGate}} s_c^t$$

$$b_{\text{OutGate}}^t = f(a_{\text{OutGate}}^t)$$

6.计算细胞输出：

$$b_c^t = b_{\text{OutGate}}^t h(s_c^t)$$

LSTM 的反向传播过程也和 BP 算法相同，其实就是一个链式求导过程，如下所示。但由于 LSTM 的结构比较复杂，因此反向传播时所要计算的梯度较多，需要读者仔细且耐心。

LSTM 的反向传播过程。

字符说明：a_{InGate}^t，a_{FGate}^t，a_{OutGate}^t 分别表示 t 时刻输入门，遗忘门，输出门的累计值；$f(x)$，$g(x)$，$h(x)$ 为激活函数；s 表示细胞的记忆信息；c 表示 LSTM 中的记忆细胞数量；I 表示输入层数量，H 表示隐藏层数量。ε_c^t，ε_s^t，$\delta_{\text{OutGate}}^t$，$\delta_{\text{InGate}}^t$，$\delta_{\text{FGate}}^t$，$\delta_c^t$ 分别表示 t 时刻：细胞输出梯度，细胞状态梯度，输出门梯度，输入门梯度，遗忘门梯度，输入单元梯度。

LSTM 各单元梯度分别为：

细胞输出梯度：

$$\varepsilon_c^t = \sum_{k=1}^{K} w_{ck} \delta_k^t + \sum_{h=1}^{H} w_{ch} \delta_h^{t+1}$$

输出门梯度：

$$\delta_{\text{OutGate}}^t = f'(a_{\text{OutGate}}^t) \sum_{c=1}^{C} h(s_c^t) \varepsilon_c^t$$

细胞状态梯度：

$$\varepsilon_s^t = b_{\text{OutGate}}^t h'(s_c^t) \varepsilon_c^t + b_{\text{FGate}}^{t+1} \varepsilon_s^{t+1} + w_{c\text{InGate}} \delta_{\text{InGate}}^{t+1} + w_{c\text{FGate}} \delta_{\text{FGate}}^{t+1} + w_{c\text{OutGate}} \delta_{\text{OutGate}}^{t+1}$$

输入单元梯度：

$$\delta_c^t = b_{\text{InGate}}^t g'(a_{\text{Input}}^t) \varepsilon_s^t$$

遗忘门梯度：

$$\delta_{\text{FGate}}^{t} = f'(a_{\text{FGate}}^{t})\sum_{c=1}^{C} s_c^{t-1}\varepsilon_c^{t}$$

输入门梯度：

$$\delta_{\text{InGate}}^{t} = f'(a_{\text{InGate}}^{t})\sum_{c=1}^{C} g(a_{\text{Input}}^{t})\varepsilon_s^{t}$$

7.3.2　门控循环单元

LSTM 是一个非常好的模型，但其会不会有些过于复杂了呢？即使我们不进行 peephole 连接，其参数也相当于传统 RNN 的 4 倍，前馈网络的 8 倍。如此多的参数，虽然使得模型能力大大加强，但同时也使得该结构过于冗余。

LSTM 的核心思想在于“门控”的概念，如果想要记住关键的信息，那我们必须学会忽略甚至遗忘一些不重要的信息，但也许我们不需要既控制信息的输入又控制信息的输出，如果当前信息很重要，那其实已经说明过去的信息可以忽略了，而这就是**门控循环单元**（Gated Recurrent Units，GRUs）[22]的基本思想。

GRUs 使用**更新门**（update gate）及**重置门**（reset gate）进行信息的更新与重置。如式（7.4）和（7.5）所示，其更新门以及重置门和 LSTM 的门结构类似，它们的输入都是当前时间片段的输入信息以及之前隐藏层的输出信息乘以权重后进行累加，然后再放入 Sigmoid 函数中激活。

$$u_j^t = f(\sum_{i=1}^{I} w_{ij}x_i^t + \sum_{h=1}^{H} w_{hj}b_h^{t-1}) \tag{7.4}$$

$$r_j^t = f(\sum_{i=1}^{I} w_{ij}x_i^t + \sum_{h=1}^{H} w_{hj}b_h^{t-1}) \tag{7.5}$$

GRUs 隐藏层的输入和 LSTM 有些区别，如式（7.6）所示，前一时间片段的隐藏层输出需要先与重置门相乘检验是否重置之后，才乘以相应的权重进行累加求和计算。

$$a_j^t = h(\sum_{i=1}^{I} w_{ij}x_i^t + \sum_{h=1}^{H} w_{hj}r_h^t b_h^{t-1}) \tag{7.6}$$

GRUs 隐藏层的输出就比较简单，经过更新门的选择后输出即可。如式（7.7）所示，更新门输出“0”或“1”。“0”表示当前信息不输出，前一隐藏层输出替代当前信息进行输出；“1”表示过滤之前信息，使用当前信息进行输出。

$$b_j^t = (1-u_j^t)b_j^{t-1} + u_j^t a_j^t \tag{7.7}$$

7.4 RNN 编程练习

本小节我们将使用 RNN 完成**图像说明**（Image Captioning）任务。该任务需要联合 CNN 与 RNN 一同学习，CNN 用于提取图像特征，RNN 用于生成图像特征所对应的说明文字。如图 7-10 所示，在训练阶段，首先我们将图片放入卷积网络中进行特征提取，将其作为隐藏层 h0 的输入；然后将图片对应的文字描述一个单词接一个单词的输入到 RNN 中，而 RNN 的输出则是当前单词的下一个预测单词。

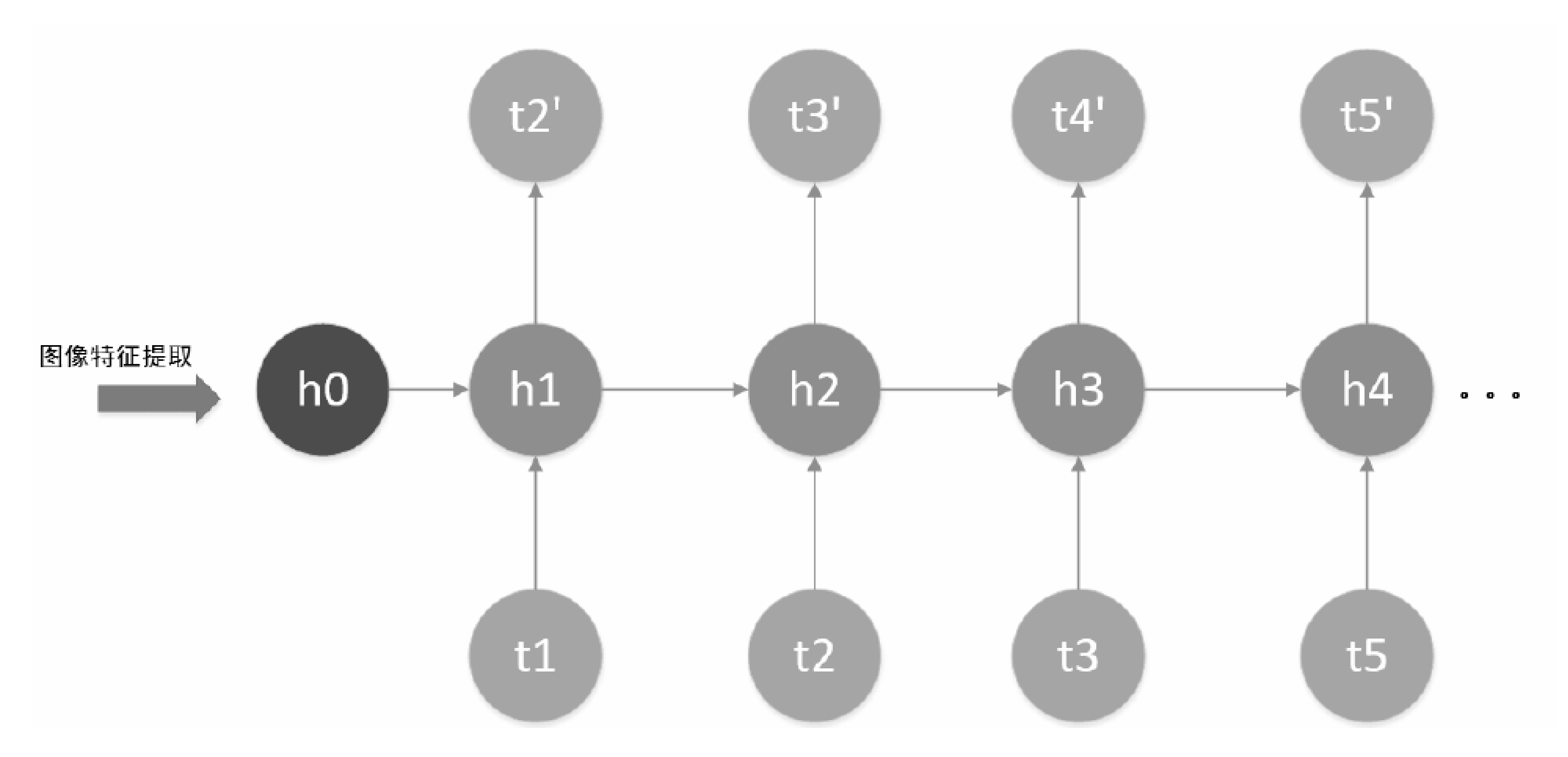


图 7-10　图片说明任务训练阶段示意图

如图 7-11 所示，在测试阶段，我们将 RNN 预测的当前单词，作为下一时间片段的输入数据输入到 RNN 中。

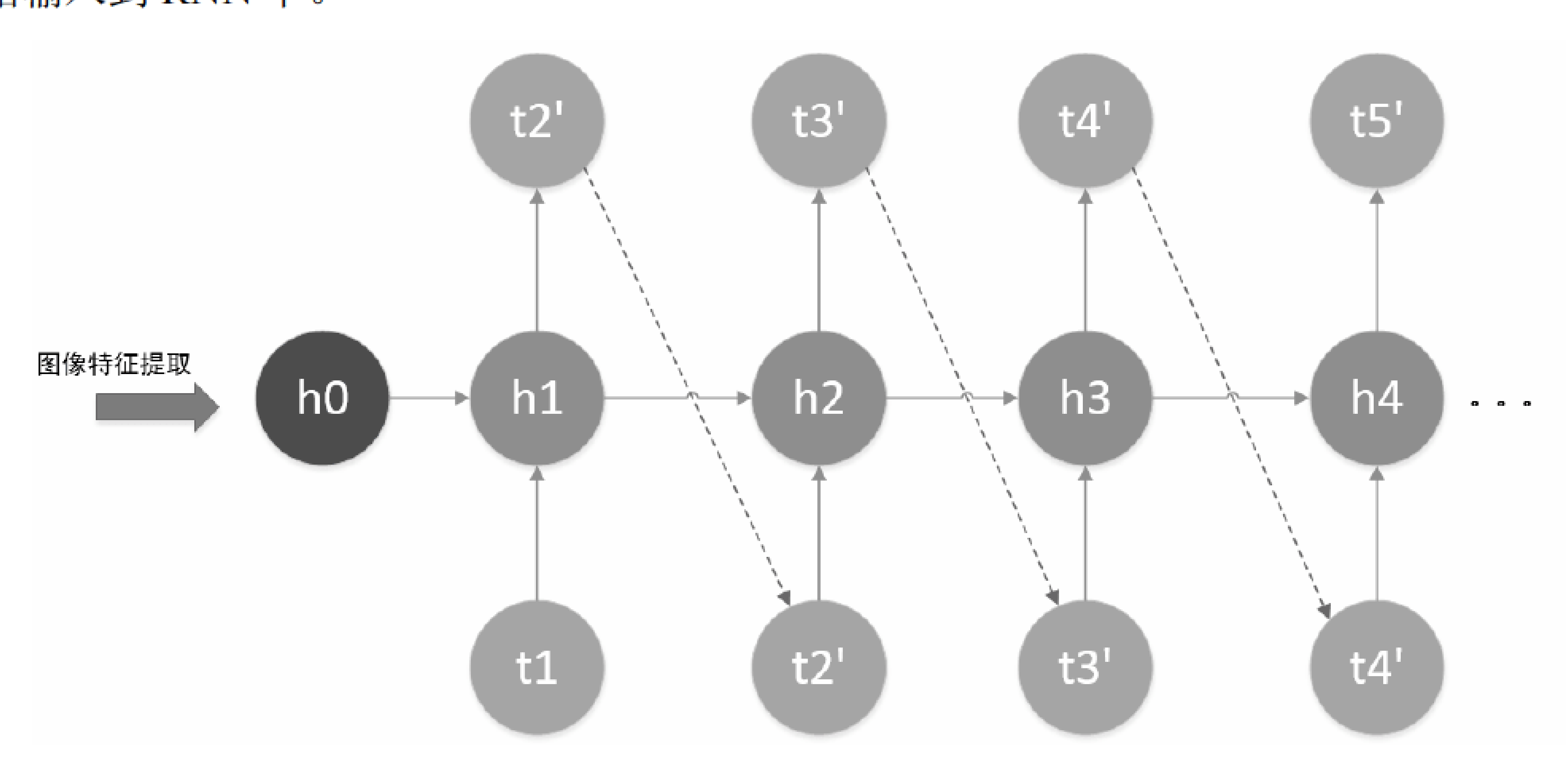


图 7-11　图片说明任务测试阶段示意图

本练习将使用 Microsoft COCO[23]数据集，其作为标准的图像说明测试数据，包含有 80000 条训练数据以及 40000 条验证数据。读者可以访问网址：http://cs231n.stanford.edu/coco_captioning.zip 来下载数据集，下载完毕后解压到 datasets 文件目录即可。该数据集已经对数据使用 VGG-16 卷积网络进行了特征提取，分别存放在 train2014_vgg16_fc7.h5 和

val2014_vgg16_fc7.h5。如果想要缩短运行时间，其也将特征从 4096 降到了 512 维，分别存放在 train2014_vgg16_fc7_pca.h5 以及 val2014_vgg16_fc7_pca.h5 中。原始图像大约有 20GB，可以使用 train2014_urls.txt 和 val2014_urls.txt 访问（需要使用 VPN）。

直接处理字符串非常不高效，因此我们给每个单词分配了字典 ID 方便我们查阅使用。因此，输入的序列数据其实是一条字典 ID 串，该文件存放在 coco2014_vocab.json。需要使用 coco_utils.py 文件中的 decode_captions 函数将 ID 转换为词向量。

在训练 RNN 前，需要使用记号<START>和<END>标记句子的开头与结束。对于一些不常用的词我们使用<UNK>进行替换。为了使句子长度相同，对于短句子，我们在<END>标记后，填充<NULL>，在训练时，不计算<NULL>的损失值与梯度。

以上内容可能过于繁杂，我们已经处理好了，你知道即可。现在打开“第 7 章练习-循环神经网络.ipnb”文件，进入本章的练习。

代码库导入模块：

```
#-*- coding: utf-8 -*-
import time, os, json
import numpy as np
import matplotlib.pyplot as plt
from classifiers.chapter7 import *
from utils import *
%matplotlib inline
plt.rcParams[ 'figure.figsize' ] = ( 10.0, 8.0 )
plt.rcParams[ 'image.interpolation' ] = 'nearest'
plt.rcParams[ 'image.cmap' ] = 'gray'
%load_ext autoreload
%autoreload 2
def rel_error( x, y ) :
  """相对误差"""
  return np.max( np.abs( x – y ) / ( np.maximum( 1e-8, np.abs( x ) + np.abs( y ) ) ) )
```

确认所有文件都存在后，接下来我们导入 COCO 数据。

数据导入代码块：

```
data = load_coco_data( pca_features = True )
for k, v in data.iteritems( ) :
  if type( v ) == np.ndarray:
    print k, type( v ), v.shape, v.dtype
  else:
    print k, type( v ), len( v )
```

正确导入数据后结果如下所示。

正确导入数据后的显示结果：

```
idx_to_word <type 'list'> 1004
train_captions <type 'numpy.ndarray'> (400135L, 17L) int32
val_captions <type 'numpy.ndarray'> (195954L, 17L) int32
train_image_idxs <type 'numpy.ndarray'> (400135L,) int32
val_features <type 'numpy.ndarray'> (40504L, 512L) float32
val_image_idxs <type 'numpy.ndarray'> (195954L,) int32
train_features <type 'numpy.ndarray'> (82783L, 512L) float32
train_urls <type 'numpy.ndarray'> (82783L,) |S63
val_urls <type 'numpy.ndarray'> (40504L,) |S63
word_to_idx <type 'dict'> 1004
```

7.4.1 RNN 单步传播

- RNN 单步前向传播

导入库文件及数据后。接下来打开“DLAction/classifiers/chapter7/rnn_layers.py”文件，完成 rnn_step_forward 函数，实现 RNN 的单步前向传播。

rnn_step_forward 函数代码块：

```
def rnn_step_forward( x, prev_h, Wx, Wh, b ):
    """
    RNN 单步前向传播，使用 tanh 激活单元。
    Inputs:
    - x: 当前时间片段数据输入( N, D )。
    - prev_h: 前一时间片段隐藏层状态 ( N, H )。
    - Wx: 输入层到隐藏层连接权重( D, H )。
    - Wh:隐藏层到隐藏层连接权重( H, H )。
    - b: 隐藏层偏置项( H, )。
    Returns 元组:
    - next_h: 下一时间片段隐藏层状态( N, H )。
    - cache: 用于反向传播的各项缓存。
    """
    next_h, cache = None, None
    ##############################################################################
    #                      任务：实现 RNN 单步前向传播。                          #
    #                    将输出值存储在 next_h 中，                              #
    #              将反向传播时所需的各项缓存存放在 cache 中。                    #
    ##############################################################################

```

```
##########################################################################
#                                结束编码                                #
##########################################################################
return next_h, cache
```

完成上述编码后运行下列代码进行检验。

```
RNN 单步前向传播代码检验：
N, D, H = 3, 10, 4
x = np.linspace( -0.4, 0.7, num = N * D ).reshape( N, D )
prev_h = np.linspace( -0.2, 0.5, num = N * H ).reshape( N, H )
Wx = np.linspace( -0.1, 0.9, num = D * H ).reshape( D, H )
Wh = np.linspace( -0.3, 0.7, num = H * H ).reshape( H, H )
b = np.linspace( -0.2, 0.4, num = H )
next_h, _ = rnn_step_forward( x, prev_h, Wx, Wh, b )
expected_next_h = np.asarray( [
  [ -0.58172089, -0.50182032, -0.41232771, -0.31410098 ],
  [ 0.66854692,   0.79562378,   0.87755553,   0.92795967 ],
  [ 0.97934501,   0.99144213,   0.99646691,   0.99854353 ] ] )
print 'next_h  误差: ', rel_error( expected_next_h, next_h )
```

```
正确编码后可能的结果：
next_h  误差:   6.29242142647e-09
```

- RNN 单步反向传播

完成 RNN 的单步前向传播后，接下来我们实现 RNN 的单步反向传播编码。打开“DLAction/classifiers/chapter7/rnn_layers.py”文件，完成 rnn_step_backward 函数，实现 RNN 的单步反向传播。

```
rnn_step_backward 函数代码块：
def rnn_step_backward( dnext_h, cache ) :
  """
  RNN 单步反向传播。
  Inputs:
  - dnext_h: 后一时间片段的梯度。
  - cache: 前向传播时的缓存。
  Returns 元组:
  - dx: 数据梯度( N, D )。
  - dprev_h: 前一时间片段梯度( N, H )。
  - dWx: 输入层到隐藏层权重梯度( D, H )。
```

```
    - dWh:   隐藏层到隐藏层权重梯度( H, H )。
    - db: 偏置项梯度( H, )。
    """
    dx, dprev_h, dWx, dWh, db = None, None, None, None, None
    ##################################################################
    #                 任务：实现 RNN 单步反向传播。                    #
    #       提示：tanh( x )梯度：  1 - tanh( x ) * tanh( x )。          #
    ##################################################################




    ##################################################################
    #                              结束编码                          #
    ##################################################################
    return dx, dprev_h, dWx, dWh, db
```

完成上述编码后运行下列代码进行检验，相对误差应该小于 1e-8。

RNN 单步反向传播梯度检验：

```
N, D, H = 4, 5, 6
x = np.random.randn( N, D )
h = np.random.randn( N, H )
Wx = np.random.randn( D, H )
Wh = np.random.randn( H, H )
b = np.random.randn( H )
out, cache = rnn_step_forward( x, h, Wx, Wh, b )
dnext_h = np.random.randn( *out.shape )
fx = lambda x: rnn_step_forward( x, h, Wx, Wh, b )[ 0 ]
fh = lambda prev_h: rnn_step_forward( x, h, Wx, Wh, b )[ 0 ]
fWx = lambda Wx: rnn_step_forward( x, h, Wx, Wh, b )[ 0 ]
fWh = lambda Wh: rnn_step_forward( x, h, Wx, Wh, b )[ 0 ]
fb = lambda b: rnn_step_forward( x, h, Wx, Wh, b )[ 0 ]
dx_num = eval_numerical_gradient_array( fx, x, dnext_h )
dprev_h_num = eval_numerical_gradient_array( fh, h, dnext_h )
dWx_num = eval_numerical_gradient_array( fWx, Wx, dnext_h )
dWh_num = eval_numerical_gradient_array( fWh, Wh, dnext_h )
db_num = eval_numerical_gradient_array( fb, b, dnext_h )
dx, dprev_h, dWx, dWh, db = rnn_step_backward( dnext_h, cache )
print 'dx  误差: ', rel_error( dx_num, dx )
```

```
print 'dprev_h  误差: ', rel_error( dprev_h_num, dprev_h )
print 'dWx  误差: ', rel_error( dWx_num, dWx )
print 'dWh  误差: ', rel_error( dWh_num, dWh )
print 'db  误差: ', rel_error( db_num, db )
```

```
梯度检验结果:
dx  误差:   1.32283815334e-10
dprev_h  误差:   9.06216381084e-11
dWx  误差:   4.70313722295e-10
dWh  误差:   6.94651543674e-10
db  误差:   2.47527388408e-10
```

7.4.2　RNN 时序传播

- RNN 前向传播

完成 RNN 的单步传播后，接下来需要将单步 RNN 组合起来，编码实现完整的时序 RNN 前向传播过程。

打开“DLAction/classifiers/chapter7/rnn_layers.py”文件，完成 rnn_forward 函数编码。

```
rnn_forward 函数代码块:
def rnn_forward( x, h0, Wx, Wh, b ) :
  """
  RNN 前向传播。
  Inputs:
  - x: 完整的时序数据( N, T, D )。
  - h0: 隐藏层初始化状态( N, H )。
  - Wx: 输入层到隐藏层权重( D, H )。
  - Wh:  隐藏层到隐藏层权重( H, H )。
  - b: 偏置项( H, )。
  Returns 元组:
  - h: 所有时间步隐藏层状态( N, T, H )。
  - cache: 反向传播所需的缓存。
  """
  h, cache = None, None
  ##################################################################
  #                     任务：实现 RNN 前向传播。                    #
  #          提示：使用前面实现的 rnn_step_forward 函数。            #
  ##################################################################
```

```
##################################################################
#                          结束编码                              #
##################################################################
return h, cache
```

完成上述编码后，使用下列代码进行验证，相对误差应该要小于 1e-7。

```
RNN 前向传播代码检验：
N, T, D, H = 2, 3, 4, 5
x = np.linspace( -0.1, 0.3, num = N * T * D ).reshape( N, T, D )
h0 = np.linspace( -0.3, 0.1, num = N * H ).reshape( N, H )
Wx = np.linspace( -0.2, 0.4, num = D * H ).reshape( D, H )
Wh = np.linspace( -0.4, 0.1, num = H * H ).reshape( H, H )
b = np.linspace( -0.7, 0.1, num = H )
h, _ = rnn_forward( x, h0, Wx, Wh, b )
expected_h = np.asarray( [
  [
    [ -0.42070749, -0.27279261, -0.11074945,   0.05740409,   0.22236251 ],
    [ -0.39525808, -0.22554661, -0.0409454,    0.14649412,   0.32397316 ],
    [ -0.42305111, -0.24223728, -0.04287027,   0.15997045,   0.35014525 ],
  ],
  [
    [ -0.55857474, -0.39065825, -0.19198182,   0.02378408,   0.23735671 ],
    [ -0.27150199, -0.07088804,   0.13562939,   0.33099728,   0.50158768 ],
    [ -0.51014825, -0.30524429, -0.06755202,   0.17806392,   0.40333043 ] ] ] )
print 'h  误差: ', rel_error( expected_h, h )
```

```
正确编码后的检验结果：
h  误差:   7.72846618019e-08
```

- RNN 时序反向传播

接下来需要将单步 RNN 反向传播组合起来，实现完整的 RNN 反向传播过程。打开“DLAction/classifiers/chapter7/rnn_layers.py”文件，完成 rnn_backward 函数编码。

```
rnn_backward 函数代码块：
def rnn_backward( dh, cache ) :
  """
```

```
RNN 反向传播。
Inputs:
- dh: 隐藏层所有时间步梯度( N, T, H )。
Returns 元组:
- dx: 输入数据时序梯度( N, T, D )。
- dh0: 初始隐藏层梯度( N, H )。
- dWx: 输入层到隐藏层权重梯度( D, H )。
- dWh: 隐藏层到隐藏层权重梯度( H, H )。
- db: 偏置项梯度( H, )。
"""
dx, dh0, dWx, dWh, db = None, None, None, None, None
##################################################################
#                    任务：实现 RNN 反向传播。                     #
#               提示：使用 rnn_step_backward 函数。                #
##################################################################








##################################################################
#                              结束编码                           #
##################################################################
return dx, dh0, dWx, dWh, db
```

完成上述编码后，使用下列代码进行梯度检验。

```
RNN 反向传播梯度检验代码块：
N, D, T, H = 2, 3, 10, 5
x = np.random.randn( N, T, D )
h0 = np.random.randn( N, H )
Wx = np.random.randn( D, H )
Wh = np.random.randn( H, H )
b = np.random.randn( H )
out, cache = rnn_forward( x, h0, Wx, Wh, b )
dout = np.random.randn( *out.shape )
dx, dh0, dWx, dWh, db = rnn_backward( dout, cache )
```

```
fx = lambda x: rnn_forward( x, h0, Wx, Wh, b )[ 0 ]
fh0 = lambda h0: rnn_forward( x, h0, Wx, Wh, b )[ 0 ]
fWx = lambda Wx: rnn_forward( x, h0, Wx, Wh, b )[ 0 ]
fWh = lambda Wh: rnn_forward( x, h0, Wx, Wh, b )[ 0 ]
fb = lambda b: rnn_forward( x, h0, Wx, Wh, b )[ 0 ]
dx_num = eval_numerical_gradient_array( fx, x, dout )
dh0_num = eval_numerical_gradient_array( fh0, h0, dout )
dWx_num = eval_numerical_gradient_array( fWx, Wx, dout )
dWh_num = eval_numerical_gradient_array( fWh, Wh, dout )
db_num = eval_numerical_gradient_array( fb, b, dout )
print 'dx  误差: ', rel_error( dx_num, dx )
print 'dh0  误差: ', rel_error( dh0_num, dh0 )
print 'dWx  误差: ', rel_error( dWx_num, dWx )
print 'dWh  误差: ', rel_error( dWh_num, dWh )
print 'db  误差: ', rel_error( db_num, db )
```

RNN 反向传播梯度检验结果：
dx 误差:　1.00522800734e-09
dh0 误差:　9.65224885793e-10
dWx 误差:　2.18316121716e-10
dWh 误差:　5.04697475273e-10
db 误差:　5.5815650463e-11

7.4.3 词嵌入

- 词嵌入前向传播

现在，需要将时序数据（索引串）转换为词向量。

打开“DLAction/classifiers/chapter7/rnn_layers.py”文件，实现 word_embedding_forward 函数，将单词索引转化为词向量。

word_embedding_forward 函数代码块：

```
def word_embedding_forward( x, W ) :
  """
  词嵌入前向传播，将数据矩阵中的 N 条长度为 T 的词索引转化为词向量。
  如：W[ x[ i, j ] ]表示第 i 条，第 j 时间步单词索引所对应的词向量。
  Inputs:
  - x: 整数型数组( N, T )，N 表示数据条数，T 表示单条数据长度，
    数组的每一元素存放着单词索引，取值范围[ 0, V )。
  - W: 词向量矩阵( V, D )存放各单词对应的向量。
```

```
Returns 元组:
- out:输出词向量( N, T, D )。
- cache:反向传播时所需的缓存。
"""
out, cache = None, None
############################################################################
#                    任务：实现词嵌入前向传播。                    #
############################################################################



############################################################################
#                             结束编码                             #
############################################################################
return out, cache
```

运行下列代码，实现的误差范围应该在 1e-8 内。

词嵌入前向传播代码检验：

```
N, T, V, D = 2, 4, 5, 3
x = np.asarray([ [ 0, 3, 1, 2 ], [ 2, 1, 0, 3 ] ] )
W = np.linspace( 0, 1, num = V * D ).reshape( V, D )
out, _ = word_embedding_forward( x, W )
expected_out = np.asarray( [
 [ [ 0.,          0.07142857,   0.14285714 ],
   [ 0.64285714,  0.71428571,   0.78571429 ],
   [ 0.21428571,  0.28571429,   0.35714286 ],
   [ 0.42857143,  0.5,          0.57142857 ] ],
 [ [ 0.42857143,  0.5,          0.57142857 ],
   [ 0.21428571,  0.28571429,   0.35714286 ],
   [ 0.,          0.07142857,   0.14285714 ],
   [ 0.64285714,  0.71428571,   0.78571429 ] ] ] )
print 'out 误差: ', rel_error( expected_out, out )
```

词嵌入前向传播检验结果：

```
out 误差:  1.00000000947e-08
```

- 词嵌入反向传播

接下来，实现 word_embedding_backward 函数，完成词嵌入的反向传播。

word_embedding_backward 函数代码块：

```
def word_embedding_backward( dout, cache ) :
    """
    词嵌入反向传播。
    Inputs:
    - dout: 上层梯度( N, T, D )。
    - cache:前向传播缓存。
    Returns:
    - dW: 词嵌入矩阵梯度( V, D )。
    """
    dW = None
    #################################################################
    #                    任务：实现词嵌入反向传播。                    #
    #                 提示：可以使用 np.add.at 函数。                 #
    #       例如 np.add.at( a, [ 1, 2 ], 1 )相当于 a[ 1 ], a[ 2 ]分别加 1。      #
    #################################################################


    #################################################################
    #                              结束编码                          #
    #################################################################
    return dW
```

运行下列梯度检验代码，误差应该小于 1e-11。

词嵌入反向传播梯度检验：

```
N, T, V, D = 50, 3, 5, 6
x = np.random.randint( V, size = ( N, T ) )
W = np.random.randn( V, D )
out, cache = word_embedding_forward( x, W )
dout = np.random.randn( *out.shape )
dW = word_embedding_backward( dout, cache )
f = lambda W: word_embedding_forward( x, W )[ 0 ]
dW_num = eval_numerical_gradient_array( f, W, dout )
print 'dW  误差: ', rel_error( dW, dW_num )
```

梯度检验结果：

```
dW  误差:   3.28045581786e-12
```

7.4.4 RNN 输出层

RNN 隐藏层到输出层的传播和前馈网络类似，只是其为(N,T,D)的三维数据。实现起来也比较简单，只需要将输入重塑为(N×T,D)数据，然后就可以进行矩阵点乘了。完整的传播过程分别在 temporal_affine_forward 和 temporal_affine_backward 函数中实现。

打开“DLAction/classifiers/chapter7/rnn_layers.py”文件，阅读相关代码。

temporal_affine_forward 函数代码块：

```
def temporal_affine_forward( x, w, b ) :
  """
  时序隐藏层仿射传播：将隐藏层时序数据( N, T, D )重塑为( N * T, D )，
  完成前向传播后，再重塑回原型输出。
  Inputs:
  - x: 时序数据( N, T, D )。
  - w: 权重( D, M )。
  - b: 偏置( M, )。
  Returns 元组:
  - out: 输出( N, T, M )。
  - cache: 反向传播缓存。
  """
  N, T, D = x.shape
  M = b.shape[ 0 ]
  out = x.reshape( N * T, D ).dot( w ).reshape( N, T, M ) + b
  cache = x, w, b, out
  return out, cache
```

temporal_affine_backward 函数代码块：

```
def temporal_affine_backward( dout, cache ) :
  """
  时序隐藏层仿射反向传播。
  Input:
  - dout:上层梯度( N, T, M )。
  - cache: 前向传播缓存。
  Returns 元组:
  - dx: 输入梯度( N, T, D )。
  - dw: 权重梯度( D, M )。
  - db: 偏置项梯度( M, )。
  """
  x, w, b, out = cache
```

```
N, T, D = x.shape
M = b.shape[ 0 ]
dx = dout.reshape( N * T, M ).dot( w.T ).reshape( N, T, D )
dw = dout.reshape( N * T, M ).T.dot( x.reshape( N * T, D ) ).T
db = dout.sum( axis = ( 0, 1 ) )
return dx, dw, db
```

阅读完上述代码，使用下列代码进行时序输出层的梯度检验。

时序输出层的梯度检验代码块：

```
N, T, D, M = 2, 3, 4, 5
x = np.random.randn( N, T, D )
w = np.random.randn( D, M )
b = np.random.randn( M )
out, cache = temporal_affine_forward( x, w, b )
dout = np.random.randn( *out.shape )
fx = lambda x: temporal_affine_forward( x, w, b )[ 0 ]
fw = lambda w: temporal_affine_forward( x, w, b )[ 0 ]
fb = lambda b: temporal_affine_forward( x, w, b )[ 0 ]
dx_num = eval_numerical_gradient_array( fx, x, dout )
dw_num = eval_numerical_gradient_array( fw, w, dout )
db_num = eval_numerical_gradient_array( fb, b, dout )
dx, dw, db = temporal_affine_backward( dout, cache )
print 'dx  误差: ', rel_error( dx_num, dx )
print 'dw  误差: ', rel_error( dw_num, dw )
print 'db  误差: ', rel_error( db_num, db )
```

时序输出梯度检验结果：

```
dx  误差:   8.73419498779e-11
dw  误差:   5.80493454126e-11
db  误差:   2.03584798919e-11
```

7.4.5　时序 Softmax 损失

使用 RNN 进行语言建模时，每一时间步都将会预测下一个单词对应单词字典中的单词得分索引。在每一时间片段，我们都将使用 Softmax 作为损失函数，然后我们再将每一时间片段的损失值加起来取平均值，但句子的长度是不相同的，为了使输入对齐，我们填充<NULL>补全短句子。这些填充的<NULL>标记不会计算在损失值或梯度值内，因此我们还需要使用 mask 进行过滤。我们已经将这些内容写进“DLAction/classifiers/chapter7/rnn_layers.py”文件中的 temporal_softmax_loss 函数，阅读即可。

temporal_softmax_loss 函数代码块：

```
def temporal_softmax_loss( x, y, mask, verbose = False ) :
    """
    时序版本的 Softmax 损失和原版本类似，只需将数据( N, T, V )重塑为( N * T, V )即可。
    需要注意的是，对于 NULL 标记不计算到损失值内，因此，需要加入掩码进行过滤。
    Inputs:
    - x:  输入数据得分( N, T, V )。
    - y:  目标索引( N, T )，其中 0 <= y[ i, t ] < V。
    - mask:  过滤 NULL 标记的掩码。
    Returns  元组:
    - loss:  损失值。
    - dx: x 梯度。
    """
    N, T, V = x.shape
    x_flat = x.reshape( N * T, V )
    y_flat = y.reshape( N * T )
    mask_flat = mask.reshape( N * T )
    probs = np.exp( x_flat - np.max( x_flat, axis = 1, keepdims = True ) )
    probs /= np.sum( probs, axis = 1, keepdims = True )
    loss = -np.sum( mask_flat * np.log( probs[ np.arange( N * T ), y_flat ] ) ) / N
    dx_flat = probs.copy( )
    dx_flat[ np.arange( N * T ), y_flat ] -= 1
    dx_flat /= N
    dx_flat *= mask_flat[ : , None ]
    if verbose: print 'dx_flat: ', dx_flat.shape
    dx = dx_flat.reshape( N, T, V )
    return loss, dx
```

阅读完时序 Softmax 的编码后，使用下列代码进行梯度检验。

时序 Softmax 梯度检验代码块：

```
N, T, V = 100, 1, 10
def check_loss( N, T, V, p ) :
    x = 0.001 * np.random.randn( N, T, V )
    y = np.random.randint( V, size = ( N, T ) )
    mask = np.random.rand( N, T ) <= p
    print temporal_softmax_loss( x, y, mask )[ 0 ]
check_loss( 100, 1, 10, 1.0 )    #  损失值大约为  2.3。
check_loss( 100, 10, 10, 1.0 )   #  损失值大约为  23。
check_loss( 5000, 10, 10, 0.1 ) #  损失值大约为  2.3。
```

```
N, T, V = 7, 8, 9
x = np.random.randn( N, T, V )
y = np.random.randint( V, size = ( N, T ) )
mask = ( np.random.rand( N, T ) > 0.5 )
loss, dx = temporal_softmax_loss( x, y, mask, verbose = False )
dx_num = eval_numerical_gradient(
    lambda x: temporal_softmax_loss( x, y, mask )[ 0 ], x, verbose = False )
print 'dx  误差: ', rel_error( dx, dx_num )
```

```
正确的检验结果：
2.30266073037
23.0259083466
2.32883265665
dx  误差:   8.8294311078e-08
```

7.4.6 RNN 图片说明任务

我们已经实现了所需的各项零件，接下来就用 RNN 完成图片说明任务。打开“DLAction/classifiers/chapter7/rnn.py”文件，阅读 CaptioningRNN 类。需要实现 RNN 的 loss 函数及测试阶段使用的 sample 函数。目前你只需实现 RNN 即可，之后再实现 LSTM 部分内容。

```
CaptioningRNN 损失函数：
def loss( self, features, captions ) :
    """
    计算 RNN 或 LSTM 的损失值。
    Inputs:
    - features:  输入图片特征( N, D )。
    - captions:  图像文字说明( N, T )。
    Returns  元组:
    - loss:  损失值。
    - grads:梯度。
    """
    #  将文字切分为两段：captions_in 除去最后一词用于 RNN 输入。
    # captions_out 除去第一个单词，用于 RNN 输出配对。
    captions_in = captions[ : , : -1 ]
    captions_out = captions[ : , 1 : ]
    #  掩码。
    mask = ( captions_out != self._null )
    #  图像仿射转换矩阵。
```

```
    W_proj, b_proj = self.params[ 'W_proj' ], self.params[ 'b_proj' ]
    # 词嵌入矩阵。
    W_embed = self.params[ 'W_embed' ]
    # RNN 参数。
    Wx, Wh, b = self.params[ 'Wx' ], self.params[ 'Wh' ], self.params[ 'b' ]
    # 隐藏层输出转化矩阵。
    W_vocab, b_vocab = self.params[ 'W_vocab' ], self.params[ 'b_vocab' ]
    loss, grads = 0.0, { }
    ########################################################################
    #             任务：实现 CaptioningRNN 传播。                           #
    # （1）使用仿射变换( features, W_proj, b_proj )，                        #
    #           将图片特征输入进隐藏层初始状态 h0( N, H )。                  #
    # （2）使用词嵌入层将 captions_in 中的单词索引转换为词向量( N, T, W )。    #
    # （3）使用 RNN 或 LSTM 处理词向量( N, T, H )。                          #
    # （4）使用时序仿射传播 temporal_affine_forward 计算各单词得分( N, T, V )。 #
    # （5）使用 temporal_softmax_loss 计算损失值。                           #
    ########################################################################









    ########################################################################
    #                            结束编码                                   #
    ########################################################################
    return loss, grads
```

测试阶段采样输出代码块：

```
def sample( self, features, max_length = 30 ) :
    """
    测试阶段的前向传播过程，采样一批图片说明作为输入。
    Inputs:
    - features:  图片特征( N, D )。
    - max_length:生成说明文字的最大长度。
```

```
Returns:
- captions: 说明文字的字典索引串( N, max_length )。
"""
N = features.shape[ 0 ]
captions = self._null * np.ones( ( N, max_length ), dtype = np.int32 )
W_proj, b_proj = self.params[ 'W_proj' ], self.params[ 'b_proj' ]
W_embed = self.params[ 'W_embed' ]
Wx, Wh, b = self.params[ 'Wx' ], self.params[ 'Wh' ], self.params[ 'b' ]
W_vocab, b_vocab = self.params[ 'W_vocab' ], self.params[ 'b_vocab' ]
######################################################################
#                    任务：测试阶段前向传播。                           #
# 提示：（1）第一个单词应该是<START>标记，captions[ : , 0 ] = self._start。 #
#       （2）当前的单词输入为前一时间段 RNN 的输出，                       #
#       （3）前向传播过程为预测当前单词的下一个单词，                       #
#            需要计算所有单词得分，然后选取最大得分作为预测单词。           #
#       （4）无法使用 rnn_forward 或 stm_forward 函数，                   #
#            需要循环调用 rnn_step_forward 或 lstm_step_forward 函数。     #
######################################################################










######################################################################
#                              结束编码                                #
######################################################################
return captions
```

完成上述编码后运行下列代码，损失误差应该小于 1e-10。

```
CaptioningRNN 损失误差检验模块：
N, D, W, H = 10, 20, 30, 40
word_to_idx = { '<NULL>': 0, 'cat': 2, 'dog': 3 }
V = len( word_to_idx )
```

```
T = 13
model = CaptioningRNN( word_to_idx, input_dim = D, wordvec_dim = W, hidden_dim = H, cell_type = 'rnn' )
for k, v in model.params.iteritems( ) :
  model.params[ k ] = np.linspace( -1.4, 1.3, num = v.size ).reshape( *v.shape )
features = np.linspace( -1.5, 0.3, num = ( N * D ) ).reshape( N, D )
captions = ( np.arange( N * T ) % V ).reshape( N, T )
loss, grads = model.loss( features, captions )
expected_loss = 9.83235591003
print '损失: ', loss
print '期望损失: ', expected_loss
print '误差: ', abs( loss - expected_loss )
```

损失误差代码检验结果：

```
损失:   9.83235591003
期望损失:   9.83235591003
误差:   2.61302091076e-12
```

梯度检验代码块：

```
batch_size = 2
timesteps = 3
input_dim = 4
wordvec_dim = 5
hidden_dim = 6
word_to_idx = { '<NULL>': 0, 'cat': 2, 'dog': 3 }
vocab_size = len( word_to_idx )
captions = np.random.randint( vocab_size, size = ( batch_size, timesteps ) )
features = np.random.randn( batch_size, input_dim )
model = CaptioningRNN( word_to_idx, input_dim = input_dim,
 wordvec_dim = wordvec_dim,
hidden_dim = hidden_dim, cell_type = 'rnn' )
loss, grads = model.loss( features, captions )
for param_name in sorted( grads ) :
  f = lambda _: model.loss( features, captions )[ 0 ]
  param_grad_num = eval_numerical_gradient( f, model.params[ param_name ], verbose = False, h = 1e-6 )
  e = rel_error( param_grad_num, grads[ param_name ] )
  print '%s  相对误差: %e' % ( param_name, e )
```

期望的梯度误差结果：

```
W_embed  相对误差: 1.245927e-09
```

```
W_proj  相对误差: 9.670782e-09
W_vocab  相对误差: 6.312240e-09
Wh  相对误差: 5.279688e-09
Wx  相对误差: 2.935529e-06
b  相对误差: 4.846914e-09
b_proj  相对误差: 9.691756e-10
b_vocab  相对误差: 3.245528e-11
```

- 过拟合数据检验

RNN 少量数据上训练损失情况如图 7-12 所示。

过拟合训练代码模块：

```
small_data = load_coco_data( max_train = 50 )
small_rnn_model = CaptioningRNN( cell_type = 'rnn', word_to_idx = data[ 'word_to_idx' ],
                                 input_dim = data[ 'train_features' ].shape[ 1 ],
                                 hidden_dim = 512, wordvec_dim = 256, )
small_rnn_solver = CaptioningTrainer( small_rnn_model, small_data,
                         update_rule = 'adam', num_epochs = 50, batch_size = 25,
                         updater_config = { 'learning_rate': 5e-3, }, lr_decay = 0.95,
                         verbose = True, print_every = 10, )
small_rnn_solver.train( )
# 绘制训练损失。
plt.plot( small_rnn_solver.loss_history )
plt.xlabel( 'Iteration' )
plt.ylabel( 'Loss' )
plt.title( 'Training loss history' )
plt.show( )
```

损失值变化情况：

```
(Iteration 1 / 100) loss: 77.138842
(Iteration 11 / 100) loss: 26.276039
(Iteration 21 / 100) loss: 7.757745
(Iteration 31 / 100) loss: 1.367408
(Iteration 41 / 100) loss: 0.532885
(Iteration 51 / 100) loss: 0.239287
(Iteration 61 / 100) loss: 0.186712
(Iteration 71 / 100) loss: 0.156235
(Iteration 81 / 100) loss: 0.148110
(Iteration 91 / 100) loss: 0.123265
```

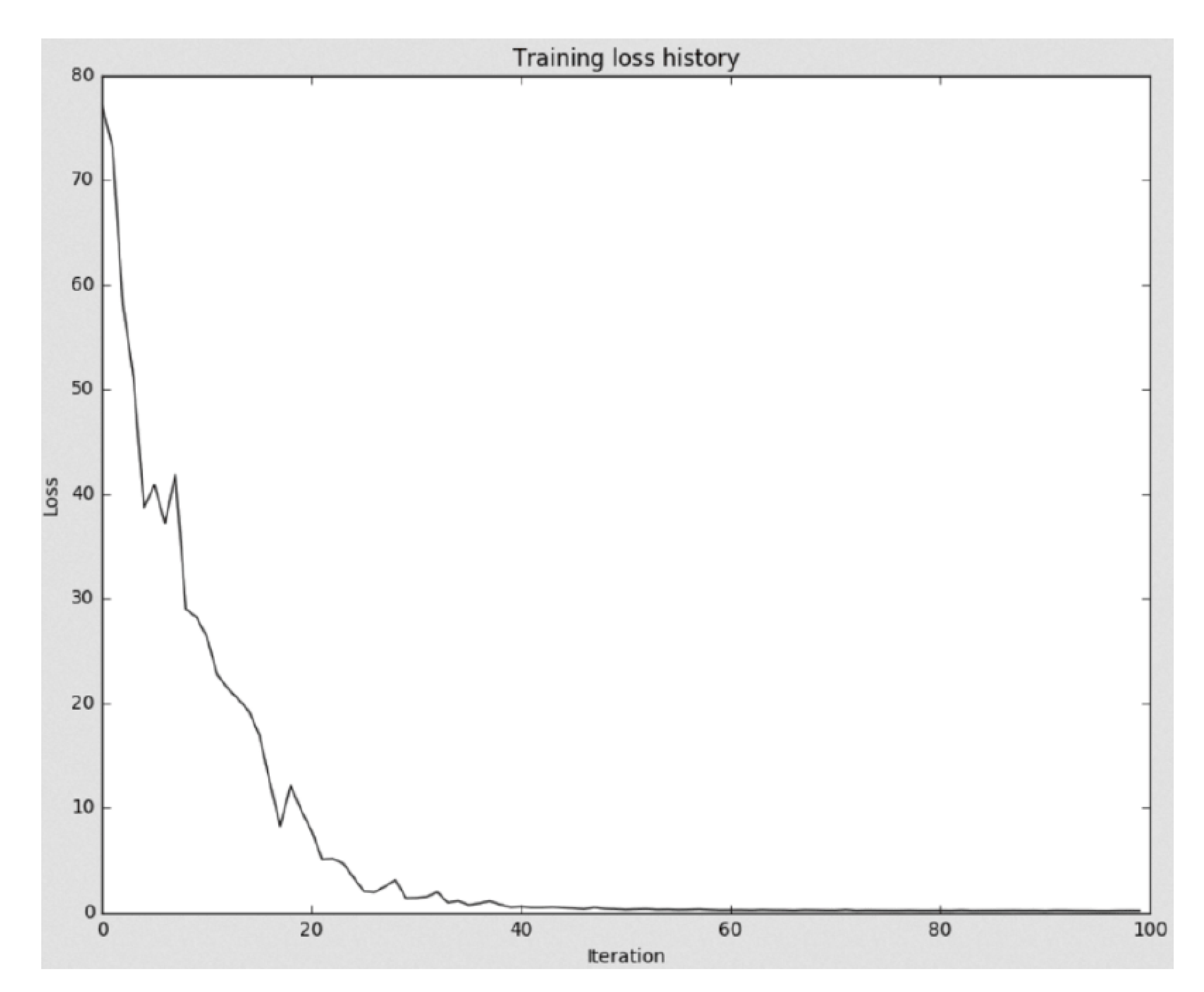


图 7-12　RNN 少量数据上训练损失情况

7.5　LSTM 编程练习

LSTM 是实践中效果最好的循环网络之一，但其传播过程可能会有些烦琐，在编程时需要仔细且耐心。接下来。我们就开始 LSTM 的编码工作。

7.5.1　LSTM 单步传播

- 单步前向传播

打开“DLAction/classifiers/chapter7/rnn_layers.py”文件，完成 lstm_step_forward 函数代码，实现 LSTM 的单步前向传播。

lstm_step_forward 函数代码块：

```
def lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b ) :
    """
    LSTM 单步前向传播。
    Inputs:
    - x: 输入数据 ( N, D )。
    - prev_h: 前一隐藏层状态 ( N, H )。
    - prev_c: 前一细胞状态( N, H )。
    - Wx: 输入层到隐藏层权重( D, 4H )。
    - Wh: 隐藏层到隐藏层权重 ( H, 4H )。
```

```
- b: 偏置项( 4H, )。
Returns 元组:
- next_h:   下一隐藏层状态( N, H )。
- next_c:   下一细胞状态( N, H )。
- cache: 反向传播所需的缓存。
"""
next_h, next_c, cache = None, None, None
##########################################################################
#              任务：实现 LSTM 单步前向传播。                              #
#        提示：稳定版本的 sigmoid 函数已经帮你实现，直接调用即可。          #
#                    tanh 函数使用 np.tanh。                              #
##########################################################################




##########################################################################
#                              结束编码                                   #
##########################################################################
return next_h, next_c, cache
```

运行下列代码，实现的误差应该在 1e-8 以内。

lstm_step_forward 函数代码检验模块：

```
N, D, H = 3, 4, 5
x = np.linspace( -0.4, 1.2, num = N * D ).reshape( N, D )
prev_h = np.linspace( -0.3, 0.7, num = N * H ).reshape( N, H )
prev_c = np.linspace( -0.4, 0.9, num = N * H ).reshape( N, H )
Wx = np.linspace( -2.1, 1.3, num = 4 * D * H ).reshape( D, 4 * H )
Wh = np.linspace( -0.7, 2.2, num = 4 * H * H ).reshape( H, 4 * H )
b = np.linspace( 0.3, 0.7, num = 4 * H )
next_h, next_c, cache = lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )
expected_next_h = np.asarray( [
    [ 0.24635157,  0.28610883,  0.32240467,  0.35525807,  0.38474904 ],
    [ 0.49223563,  0.55611431,  0.61507696,  0.66844003,  0.7159181 ],
    [ 0.56735664,  0.66310127,  0.74419266,  0.80889665,  0.858299   ] ] )
expected_next_c = np.asarray( [
    [ 0.32986176,  0.39145139,  0.451556,    0.51014116,  0.56717407 ],
    [ 0.66382255,  0.76674007,  0.87195994,  0.97902709,  1.08751345 ],
    [ 0.74192008,  0.90592151,  1.07717006,  1.25120233,  1.42395676 ] ] )
```

```
print 'next_h 误差: ', rel_error( expected_next_h, next_h )
print 'next_c 误差: ', rel_error( expected_next_c, next_c )
```

lstm_step_forward 代码检验结果：
next_h 误差: 5.70541311858e-09 next_c 误差: 5.81431230888e-09

- LSTM 单步反向传播

打开“DLAction/classifiers/chapter7/rnn_layers.py”文件，完成 lstm_step_backward 函数代码，实现 LSTM 的单步反向传播。

lstm_step_backward 函数代码块：

```
def lstm_step_backward( dnext_h, dnext_c, cache ) :
  """
   LSTM 单步反向传播。
  Inputs:
  - dnext_h: 下一隐藏层梯度( N, H )。
  - dnext_c: 下一细胞梯度( N, H )。
  - cache: 前向传播缓存。
  Returns 元组:
  - dx: 输入数据梯度( N, D )。
  - dprev_h: 前一隐藏层梯度( N, H )。
  - dprev_c: 前一细胞梯度( N, H )。
  - dWx: 输入层到隐藏层梯度( D, 4H )。
  - dWh: 隐藏层到隐藏层梯度( H, 4H )。
  - db: 偏置梯度( 4H, )。
  """
  dx, dprev_h, dc, dWx, dWh, db = None, None, None, None, None, None
  ##########################################################################
  #                         任务：实现 LSTM 单步反向传播。                  #
  #      提示：sigmoid( x )函数梯度：sigmoid( x ) * ( 1- sigmoid( x ) )       #
  #                tanh( x )函数梯度：1 - tanh( x ) * tanh( x )             #
  ##########################################################################
```

```
    ##########################################################################
    #                              结束编码                                    #
    ##########################################################################
    return dx, dprev_h, dprev_c, dWx, dWh, db
```

lstm_step_backward 梯度检验代码块：

```
N, D, H = 4, 5, 6
x = np.random.randn( N, D )
prev_h = np.random.randn( N, H )
prev_c = np.random.randn( N, H )
Wx = np.random.randn( D, 4 * H )
Wh = np.random.randn( H, 4 * H )
b = np.random.randn( 4 * H )
next_h, next_c, cache = lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )
dnext_h = np.random.randn( *next_h.shape )
dnext_c = np.random.randn( *next_c.shape )
fx_h = lambda x: lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )[ 0 ]
fh_h = lambda h: lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )[ 0 ]
fc_h = lambda c: lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )[ 0 ]
fWx_h = lambda Wx: lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )[ 0 ]
fWh_h = lambda Wh: lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )[ 0 ]
fb_h = lambda b: lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )[ 0 ]
fx_c = lambda x: lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )[ 1 ]
fh_c = lambda h: lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )[ 1 ]
fc_c = lambda c: lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )[ 1 ]
fWx_c = lambda Wx: lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )[ 1 ]
fWh_c = lambda Wh: lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )[ 1 ]
fb_c = lambda b: lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b )[ 1 ]
num_grad = eval_numerical_gradient_array
dx_num = num_grad( fx_h, x, dnext_h ) + num_grad( fx_c, x, dnext_c )
dh_num = num_grad( fh_h, prev_h, dnext_h ) + num_grad( fh_c, prev_h, dnext_c )
dc_num = num_grad( fc_h, prev_c, dnext_h ) + num_grad( fc_c, prev_c, dnext_c )
dWx_num = num_grad( fWx_h, Wx, dnext_h ) + num_grad( fWx_c, Wx, dnext_c )
dWh_num = num_grad( fWh_h, Wh, dnext_h ) + num_grad( fWh_c, Wh, dnext_c )
db_num = num_grad( fb_h, b, dnext_h ) + num_grad( fb_c, b, dnext_c )
dx, dh, dc, dWx, dWh, db = lstm_step_backward( dnext_h, dnext_c, cache )
print 'dx  误差: ', rel_error( dx_num, dx )
print 'dh  误差: ', rel_error( dh_num, dh )
```

```
print 'dc  误差: ', rel_error( dc_num, dc )
print 'dWx  误差: ', rel_error( dWx_num, dWx )
print 'dWh  误差: ', rel_error( dWh_num, dWh )
print 'db  误差: ', rel_error( db_num, db )
```

梯度检验结果：

```
dx  误差:   2.24750611741e-10
dh  误差:   8.31744482897e-10
dc  误差:   1.1349577207e-09
dWx  误差:   2.02890276008e-09
dWh  误差:   7.78920750327e-09
db  误差:   1.29574707381e-09
```

7.5.2　LSTM 时序传播

- LSTM 前向传播

现在将单步前向传播组合起来，完成完整的时序传播。打开“DLAction/classifiers/chapter7/rnn_layers.py”文件，完成 lstm_forward 函数代码，实现 LSTM 的前向传播。

lstm_forward 函数代码块：

```
def lstm_forward( x, h0, Wx, Wh, b ) :
    """
    LSTM 前向传播。
    Inputs:
    - x:  输入数据 ( N, T, D )。
    - h0:初始化隐藏层状态( N, H )。
    - Wx:  输入层到隐藏层权重 ( D, 4H )。
    - Wh:  隐藏层到隐藏层权重( H, 4H )。
    - b:  偏置项( 4H, )。
    Returns  元组:
    - h:  隐藏层所有状态 ( N, T, H )。
    - cache:  用于反向传播的缓存。
    """
    h, cache = None, None
    ###########################################################################
    #                          任务：实现完整的 LSTM 前向传播。                  #
    ###########################################################################
```

```
    ##################################################################
    #                          结束编码                               #
    ##################################################################
    return h, cache
```

运行下列代码，实现误差应该在 1e-7 以内。

lstm_forward 函数代码检验模块：

```
N, D, H, T = 2, 5, 4, 3
x = np.linspace( -0.4, 0.6, num = N * T * D ).reshape( N, T, D )
h0 = np.linspace( -0.4, 0.8, num = N * H ).reshape( N, H )
Wx = np.linspace( -0.2, 0.9, num = 4 * D * H ).reshape( D, 4 * H )
Wh = np.linspace( -0.3, 0.6, num = 4 * H * H ).reshape( H, 4 * H )
b = np.linspace( 0.2, 0.7, num = 4 * H )
h, cache = lstm_forward( x, h0, Wx, Wh, b )
expected_h = np.asarray( [
 [ [ 0.01764008,   0.01823233,   0.01882671,   0.0194232 ],
   [ 0.11287491,   0.12146228,   0.13018446,   0.13902939 ],
   [ 0.31358768,   0.33338627,   0.35304453,   0.37250975 ] ],
 [ [ 0.45767879,   0.4761092,    0.4936887,    0.51041945 ],
   [ 0.6704845,    0.69350089,   0.71486014,   0.7346449 ],
   [ 0.81733511,   0.83677871,   0.85403753,   0.86935314 ] ] ] )
print 'h  误差: ', rel_error( expected_h, h )
```

期望的检验结果：

```
h  误差:   8.61053745211e-08
```

- LSTM 反向传播

现在将单步反向传播组合起来，完成完整的时序反向传播。打开"DLAction/classifiers/chapter7/rnn_layers.py"文件，完成 lstm_backward 函数代码，实现 LSTM 的反向传播。

lstm_backward 函数代码块：

```
def lstm_backward( dh, cache ) :
    """
    LSTM 反向传播。
    Inputs:
```

```
- dh: 各隐藏层梯度( N, T, H )。
- cache: V 前向传播缓存。
Returns 元组:
- dx: 输入数据梯度 ( N, T, D )。
- dh0:初始隐藏层梯度( N, H )。
- dWx: 输入层到隐藏层权重梯度 ( D, 4H )。
- dWh: 隐藏层到隐藏层权重梯度 ( H, 4H )。
- db: 偏置项梯度 ( 4H, )。
"""
dx, dh0, dWx, dWh, db = None, None, None, None, None
###############################################################
#                   任务：实现完整的 LSTM 反向传播。            #
###############################################################




###############################################################
#                            结束编码                          #
###############################################################
return dx, dh0, dWx, dWh, db
```

运行下列代码，误差大约在 1e-8。

LSTM 反向传播梯度检验代码块：

```
N, D, T, H = 2, 3, 10, 6
x = np.random.randn( N, T, D )
h0 = np.random.randn( N, H )
Wx = np.random.randn( D, 4 * H )
Wh = np.random.randn( H, 4 * H )
b = np.random.randn( 4 * H )
out, cache = lstm_forward( x, h0, Wx, Wh, b )
dout = np.random.randn( *out.shape )
dx, dh0, dWx, dWh, db = lstm_backward( dout, cache )
fx = lambda x: lstm_forward( x, h0, Wx, Wh, b )[ 0 ]
fh0 = lambda h0: lstm_forward( x, h0, Wx, Wh, b )[ 0 ]
fWx = lambda Wx: lstm_forward( x, h0, Wx, Wh, b )[ 0 ]
fWh = lambda Wh: lstm_forward( x, h0, Wx, Wh, b )[ 0 ]
fb = lambda b: lstm_forward( x, h0, Wx, Wh, b )[ 0 ]
dx_num = eval_numerical_gradient_array( fx, x, dout )
```

```
dh0_num = eval_numerical_gradient_array( fh0, h0, dout )
dWx_num = eval_numerical_gradient_array( fWx, Wx, dout )
dWh_num = eval_numerical_gradient_array( fWh, Wh, dout )
db_num = eval_numerical_gradient_array( fb, b, dout )
print 'dx  误差: ', rel_error( dx_num, dx )
print 'dh0  误差: ', rel_error( dx_num, dx )
print 'dWx  误差: ', rel_error( dx_num, dx )
print 'dWh  误差: ', rel_error( dx_num, dx )
print 'db  误差: ', rel_error( dx_num, dx )
```

```
梯度检验结果：
dx  误差:   9.59964787707e-10
dh0  误差:   9.59964787707e-10
dWx  误差:   9.59964787707e-10
dWh  误差:   9.59964787707e-10
db  误差:   9.59964787707e-10
```

7.5.3 LSTM 实现图片说明任务

打开“DLAction/classifiers/chapter7/rnn.py”文件，完成 LSTM 模式下的损失计算。由于之前已经完成了 RNN 模式，现在只需要添加少量代码即可。完成编码后运行下列代码。

```
LSTM 模式代码检验：
N, D, W, H = 10, 20, 30, 40
word_to_idx = { '<NULL>': 0, 'cat': 2, 'dog': 3 }
V = len( word_to_idx )
T = 13
model = CaptioningRNN( word_to_idx, input_dim = D, wordvec_dim = W, hidden_dim = H, cell_type = 'lstm' )
for k, v in model.params.iteritems( ) :
    model.params[ k ] = np.linspace( -1.4, 1.3, num = v.size ).reshape( *v.shape )
features = np.linspace( -0.5, 1.7, num = N * D ).reshape( N, D )
captions = ( np.arange( N * T ) % V ).reshape( N, T )
loss, grads = model.loss( features, captions )
expected_loss = 9.82445935443
print '损失值: ', loss
print '期望损失值: ', expected_loss
print '误差: ', abs( loss - expected_loss )
```

LSTM 损失值检验结果：
损失值:　9.82445935443
期望损失值:　9.82445935443
误差:　2.26485497024e-12

- LSTM 过拟合测试

和 RNN 类似，我们测试 LSTM 能否在小数据集上过拟合。LSTM 少量数据上训练损失情况如图 7-13 所示。

LSTM 过拟合测试代码块：

```
small_data = load_coco_data( max_train = 50 )
small_lstm_model = CaptioningRNN( cell_type = 'lstm',
word_to_idx = data[ 'word_to_idx' ],
   input_dim = data[ 'train_features' ].shape[ 1 ],
hidden_dim = 512, wordvec_dim = 256 )
small_lstm_solver = CaptioningTrainer( small_lstm_model, small_data, update_rule = 'adam',
                                   num_epochs = 50, batch_size = 25,
                                 updater_config = { 'learning_rate': 5e-3, },
                                 lr_decay=0.995, verbose = True, print_every = 10, )
small_lstm_solver.train( )
plt.plot( small_lstm_solver.loss_history )
plt.xlabel( 'Iteration' )
plt.ylabel( 'Loss' )
plt.title( 'Training loss history' )
plt.show( )
```

训练损失值变化结果：

```
(Iteration 1 / 100) loss: 80.749666
(Iteration 11 / 100) loss: 49.919856
(Iteration 21 / 100) loss: 27.695865
(Iteration 31 / 100) loss: 17.968135
(Iteration 41 / 100) loss: 8.942594
(Iteration 51 / 100) loss: 2.322836
(Iteration 61 / 100) loss: 1.350197
(Iteration 71 / 100) loss: 0.482612
(Iteration 81 / 100) loss: 0.353356
(Iteration 91 / 100) loss: 0.185770
```

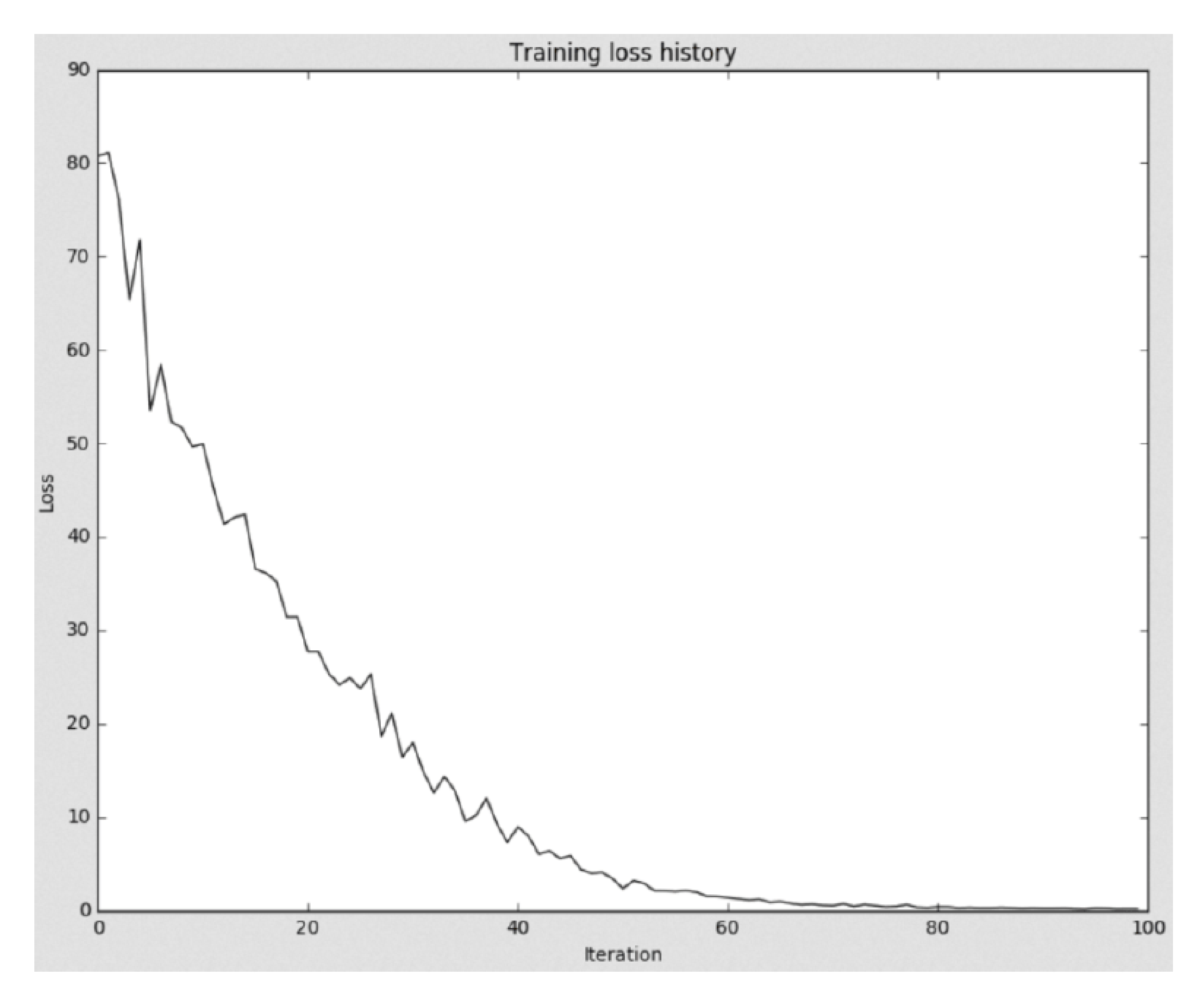


图 7-13　LSTM 少量数据上训练损失情况

7.6　参考代码

7.6.1　RNN 参考代码

rnn_step_forward 函数代码块：

```
def rnn_step_forward( x, prev_h, Wx, Wh, b ) :
    next_h, cache = None, None
    a = prev_h.dot( Wh ) + x.dot( Wx ) + b
    next_h = np.tanh( a )
    cache = ( x, prev_h, Wh, Wx, b, next_h )
    return next_h, cache
```

rnn_step_backward 函数代码块：

```
def rnn_step_backward( dnext_h, cache ) :
    dx, dprev_h, dWx, dWh, db = None, None, None, None, None
    x, prev_h, Wh, Wx, b, next_h = cache
    dscores = dnext_h * ( 1 - next_h * next_h )
    dWx = np.dot( x.T, dscores )
    db = np.sum( dscores, axis = 0 )
    dWh = np.dot( prev_h.T, dscores )
```

```
    dx = np.dot( dscores, Wx.T )
    dprev_h = np.dot( dscores, Wh.T )
    return dx, dprev_h, dWx, dWh, db
```

rnn_forward 函数代码块：

```
def rnn_forward( x, h0, Wx, Wh, b ) :
    h, cache = None, None
    N, T, D = x.shape
    ( H , ) = b.shape
    h = np.zeros( ( N, T, H ) )
    prev_h = h0
    for t in range( T ) :
        xt = x[ : , t, : ]
        next_h, _ = rnn_step_forward( xt, prev_h, Wx, Wh, b )
        prev_h = next_h
        h[ : , t, : ] = prev_h
    cache = ( x, h0, Wh, Wx, b, h )
    return h, cache
```

rnn_backward 函数代码块：

```
def rnn_backward( dh, cache ) :
    dx, dh0, dWx, dWh, db = None, None, None, None, None
    x, h0, Wh, Wx, b, h = cache
    N, T, H = dh.shape
    _, _, D = x.shape
    next_h = h[ : ,T – 1 , : ]
    dprev_h = np.zeros( ( N, H ) )
    dx = np.zeros( ( N, T, D ) )
    dh0 = np.zeros( ( N, H ) )
    dWx= np.zeros( ( D, H ) )
    dWh = np.zeros( ( H, H ) )
    db = np.zeros( ( H, ) )
    for t in range( T ) :
        t = T – 1 – t
        xt = x[ : , t , : ]
        if t ==0 :
                prev_h = h0
        else:
                prev_h = h[ : , t – 1 , : ]
```

```
        step_cache = ( xt, prev_h, Wh, Wx, b, next_h )
        next_h = prev_h
        dnext_h = dh[ : ,t , : ] + dprev_h
        dx[ : , t , : ], dprev_h, dWxt, dWht, dbt = rnn_step_backward( dnext_h, step_cache )
        dWx, dWh, db = dWx + dWxt, dWh + dWht, db + dbt
    dh0 = dprev_h
    return dx, dh0, dWx, dWh, db
```

word_embedding_forward 函数代码块：

```
def word_embedding_forward( x, W ) :
    out, cache = None, None
    N, T = x.shape
    V, D = W.shape
    out = np.zeros( ( N, T, D ) )
    for i in range( N ) :
        for j in range( T ) :
            out[ i, j ] = W[ x[ i, j ] ]
    cache = ( x, W.shape )
    return out, cache
```

word_embedding_backward 函数代码块：

```
def word_embedding_backward( dout, cache ) :
    dW = None
    x, W_shape = cache
    dW = np.zeros( W_shape )
    np.add.at( dW, x, dout )
    return dW
```

CaptioningRNN 损失函数：

```
def loss( self, features, captions ) :
        captions_in = captions[ : , : -1 ]
        captions_out = captions[ : , 1 : ]
        #  掩码
        mask = ( captions_out != self._null )
        #  图像仿射转换矩阵
        W_proj, b_proj = self.params[ 'W_proj' ], self.params[ 'b_proj' ]
        #  词嵌入矩阵
        W_embed = self.params[ 'W_embed' ]
        # RNN 参数
```

```
Wx, Wh, b = self.params[ 'Wx' ], self.params[ 'Wh' ], self.params[ 'b' ]
# 隐藏层输出转化矩阵
W_vocab, b_vocab = self.params[ 'W_vocab' ], self.params[ 'b_vocab' ]
loss, grads = 0.0, { }
# ( 1 )
h0, cache_h0 = affine_forward( features, W_proj, b_proj )
# ( 2 )
x, cache_embedding = word_embedding_forward( captions_in, W_embed )
# ( 3 )
if self.cell_type == 'rnn':
    out_h, cache_rnn = rnn_forward( x, h0, Wx, Wh, b )
elif self.cell_type == 'lstm':
    out_h, cache_rnn = lstm_forward( x, h0, Wx, Wh, b )
else:
    raise ValueError( 'Invalid cell_type "%s" ' % self.cell_type )
# ( 4 )
yHat, cache_out = temporal_affine_forward( out_h, W_vocab, b_vocab )
# ( 5 )
loss, dy = temporal_softmax_loss( yHat, captions_out, mask, verbose = False )
# 计算梯度
dout_h, dW_vocab, db_vocab = temporal_affine_backward( dy, cache_out )
if self.cell_type == 'rnn':
    dx, dh0, dWx, dWh, db = rnn_backward( dout_h, cache_rnn )
elif self.cell_type == 'lstm':
    dx, dh0, dWx, dWh, db = lstm_backward( dout_h, cache_rnn )
else:
    raise ValueError( 'Invalid cell_type "%s" ' % self.cell_type )
dW_embed = word_embedding_backward( dx, cache_embedding )
dfeatures, dW_proj, db_proj = affine_backward( dh0, cache_h0 )
grads[ 'W_proj' ] = dW_proj
grads[ 'b_proj' ] = db_proj
grads[ 'W_embed' ] = dW_embed
grads[ 'Wx' ] = dWx
grads[ 'Wh' ] = dWh
grads[ 'b' ] = db
grads[ 'W_vocab' ] = dW_vocab
grads[ 'b_vocab' ] = db_vocab
return loss, grads
```

测试阶段采样输出代码块：

```
def sample( self, features, max_length = 30 ) :
  N = features.shape[ 0 ]
  captions = self._null * np.ones( ( N, max_length ), dtype = np.int32 )
  W_proj, b_proj = self.params[ 'W_proj' ], self.params[ 'b_proj' ]
  W_embed = self.params[ 'W_embed' ]
  Wx, Wh, b = self.params[ 'Wx' ], self.params[ 'Wh' ], self.params[ 'b' ]
  W_vocab, b_vocab = self.params[ 'W_vocab' ], self.params[ 'b_vocab' ]
  N, D = features.shape
  affine_out, affine_cache = affine_forward( features , W_proj, b_proj )
  prev_word_idx = [ self._start ] * N
  prev_h = affine_out
  prev_c = np.zeros( prev_h.shape )
  captions[ : , 0 ] = self._start
  for i in range( 1, max_length ) :
      prev_word_embed   = W_embed[ prev_word_idx ]
      if self.cell_type == 'rnn':
          next_h, rnn_step_cache = rnn_step_forward( prev_word_embed,prev_h, Wx, Wh, b )
      elif self.cell_type == 'lstm':
          next_h, next_c, lstm_step_cache = lstm_step_forward( prev_word_embed,
                                                               prev_h, prev_c, Wx, Wh, b )
          prev_c = next_c
      else:
          raise ValueError( 'Invalid cell_type "%s" ' % self.cell_type )
      vocab_affine_out, vocab_affine_out_cache = affine_forward( next_h, W_vocab,b_vocab )
      captions[ : , i ] = list( np.argmax( vocab_affine_out, axis = 1 ) )
      prev_word_idx = captions[ : , i ]
      prev_h = next_h
  return captions
```

7.6.2 LSTM 参考代码

lstm_step_forward 函数代码块：

```
def lstm_step_forward( x, prev_h, prev_c, Wx, Wh, b ) :
  next_h, next_c, cache = None, None, None
  N, D = x.shape
  N, H = prev_h.shape
  input_gate = sigmoid( np.dot( x, Wx[ : , 0 : H ] ) + np.dot(
                                  prev_h, Wh[ : , 0 : H ] ) + b[ 0 : H ] )
```

```
    forget_gate = sigmoid( np.dot( x, Wx[ : , H : 2 * H ] ) + np.dot(
                                        prev_h, Wh[ : , H : 2 * H ] ) + b[ H : 2 * H ] )
    output_gate = sigmoid( np.dot( x, Wx[ : , 2 * H : 3 * H ] ) + np.dot(
                                        prev_h, Wh[ : , 2 * H : 3 * H ] ) + b[ 2 * H : 3 * H ] )
    input = np.tanh( np.dot( x, Wx[ : , 3 * H : 4 * H ] ) + np.dot(
                                        prev_h, Wh[ : , 3 * H : 4 * H ] ) + b[ 3 * H : 4 * H ] )
    next_c = forget_gate * prev_c + input * input_gate
    next_scores_c = np.tanh( next_c )
    next_h = output_gate * next_scores_c
    cache = ( x, Wx, Wh, b, input, input_gate, output_gate, forget_gate, prev_h, prev_c, next_scores_c )
    return next_h, next_c, cache
```

lstm_step_backward 函数代码块:

```
def lstm_step_backward( dnext_h, dnext_c, cache ) :
    dx, dprev_h, dc, dWx, dWh, db = None, None, None, None, None, None
    x, Wx, Wh, b, input, input_gate, output_gate, forget_gate, prev_h, prev_c, next_scores_c = cache
    N, D = x.shape
    N, H = prev_h.shape
    dWx = np.zeros( ( D, 4 * H ) )
    dxx = np.zeros( ( D, 4 * H ) )
    dWh = np.zeros( ( H, 4 * H ) )
    dhh = np.zeros( ( H, 4 * H ) )
    db = np.zeros( 4 * H )
    dx =np.zeros( ( N, D ) )
    dprev_h = np.zeros( ( N, H ) )
    dc_tem = dnext_c + dnext_h * ( 1 - next_scores_c **2 ) * output_gate
    dprev_c = forget_gate * dc_tem
    dforget_gate = prev_c * dc_tem
    dinput_gate = input * dc_tem
    dinput = input_gate * dc_tem
    doutput_gate = next_scores_c * dnext_h
    dscores_in_gate = input_gate * ( 1 - input_gate ) * dinput_gate
    dscores_forget_gate = forget_gate * ( 1 - forget_gate ) * dforget_gate
    dscores_out_gate = output_gate * ( 1 - output_gate ) * doutput_gate
    dscores_in = ( 1 – input **2 ) * dinput
    da = np.hstack( ( dscores_in_gate, dscores_forget_gate, dscores_out_gate, dscores_in ) )
    dWx = np.dot( x.T, da )
    dWh = np.dot( prev_h.T, da )
    db = np.sum( da, axis = 0 )
```

```
    dx = np.dot( da, Wx.T )
    dprev_h = np.dot( da, Wh.T )
    return dx, dprev_h, dprev_c, dWx, dWh, db
```

lstm_forward 函数代码块：

```
def lstm_forward( x, h0, Wx, Wh, b ) :
    h, cache = None, None
    N, T, D = x.shape
    H = b.shape[ 0 ] / 4
    h = np.zeros( ( N, T, H ) )
    cache = { }
    prev_h = h0
    prev_c = np.zeros( ( N, H ) )
    for t in range( T ) :
        xt = x[ : , t, : ]
        next_h, next_c, cache[ t ] = lstm_step_forward( xt, prev_h, prev_c, Wx, Wh, b )
        prev_h = next_h
        prev_c = next_c
        h[ : , t, : ] = prev_h
    return h, cache
```

lstm_backward 函数代码块：

```
def lstm_backward( dh, cache ) :
    dx, dh0, dWx, dWh, db = None, None, None, None, None
    N, T, H = dh.shape
    x, Wx, Wh, b, input, input_gate, output_gate, forget_gate, prev_h, prev_c, next_scores_c = cache[ T – 1 ]
    D = x.shape[ 1 ]
    dprev_h = np.zeros( ( N, H ) )
    dprev_c = np.zeros( ( N, H ) )
    dx = np.zeros( ( N, T, D ) )
    dh0 = np.zeros( ( N, H ) )
    dWx= np.zeros( ( D, 4 * H ) )
    dWh = np.zeros( ( H, 4 * H ) )
    db = np.zeros( ( 4 * H, ) )
    for t in range( T ) :
        t = T – 1 – t
        step_cache = cache[ t ]
        dnext_h = dh[ : ,t , : ] + dprev_h
        dnext_c = dprev_c
```

```
    dx[ : , t , : ], dprev_h, dprev_c, dWxt, dWht, dbt = lstm_step_backward(dnext_h, dnext_c, step_cache )
    dWx, dWh, db = dWx + dWxt, dWh + dWht, db + dbt
  dh0 = dprev_h
  return dx, dh0, dWx, dWh, db
```

7.7 参考文献

[1] Indyk, P., & Motwani, R. (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. Paper presented at the Thirtieth ACM Symposium on Theory of Computing.

[2] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, 323(6088), 533-536.

[3] Pollack, J. B. (1990). Recursive distributed representations. Artificial Intelligence, 46(1), 77-105.

[4] Ahmad, A. M., Ismail, S., & Samaon, D. F. (2004). Recurrent neural network with backpropagation through time for speech recognition. Paper presented at the IEEE International Symposium on Communications and Information Technology.

[5] Devlin, J., Cheng, H., Fang, H., Gupta, S., Deng, L., He, X., . . . Mitchell, M. (2015). Language Models for Image Captioning: The Quirks and What Works. Computer Science.

[6] Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45(11), 2673-2681.

[7] Graves, A., & Schmidhuber, J. (2012). Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks. Paper presented at the Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December.

[8] Graves, A., Mohamed, A. R., & Hinton, G. (2013). Speech recognition with deep recurrent neural networks. 38(2003), 6645-6649.

[9] Cho, K., Merrienboer, B. V., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. Computer Science.

[10] Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. 4, 3104-3112.

[11] Hihi, S. E., Hc-J, M. Q., & Bengio, Y. (1995). Hierarchical Recurrent Neural Networks for Long-Term Dependencies. Advances in neural information processing systems, 8, 493-499.

[12] Graves, A. (2014). Generating Sequences With Recurrent Neural Networks. Computer Science.

[13] Pascanu, R., Gulcehre, C., Cho, K., & Bengio, Y. (2013). How to Construct Deep Recurrent Neural Networks. Computer Science.

[14] Pascanu, R., Mikolov, T., & Bengio, Y. (2013). On the difficulty of training recurrent

neural networks. Paper presented at the International Conference on International Conference on Machine Learning.

[15] Bengio, Y., Frasconi, P., & Simard, P. (1993). The problem of learning long-term dependencies in recurrent networks. Paper presented at the IEEE International Conference on Neural Networks.

[16] Hochreiter, S., & Schmidhuber, J. (2012). Long short-term memory. Neural Computation, 9(8), 1735-1780.

[17] Graves, A., & Jaitly, N. (2014). Towards end-to-end speech recognition with recurrent neural networks. Paper presented at the International Conference on Machine Learning.

[18] Kiros, R., Salakhutdinov, R., & Zemel, R. S. (2014). Unifying Visual-Semantic Embeddings with Multimodal Neural Language Models. Computer Science.

[19] Vinyals, O., Kaiser, L., Koo, T., Petrov, S., Sutskever, I., & Hinton, G. (2014). Grammar as a Foreign Language. Eprint Arxiv.

[20] Gers, F. A., Schmidhuber, J., & Cummins, F. (2000). Learning to Forget: Continual Prediction with LSTM. Neural Computation 12(10): 2451-2471. Neural Computation, 12(10), 2451-2471.

[21] Gers, F. A., Schraudolph, N. N., Schmidhuber, J., & rgen. (2003). Learning precise timing with lstm recurrent networks: JMLR.org.

[22] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2015). Gated Feedback Recurrent Neural Networks. Computer Science, 2067-2075.

[23] Lin, T. Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., ... Zitnick, C. L. (2015). Microsoft COCO: Common Objects in Context. 8693, 740-755.

第 8 章

TensorFlow 快速入门

在本书前面的章节中我们介绍了大量常用的深度学习技术，我们一步一步实现了 Softmax、DNN、CNN、RNN 和 LSTM 等模型。但很遗憾地告诉你，你实现的这些模型仅能作为学习演示使用，它们根本无法满足实际应用的需要，因为它们的运行速度实在是太缓慢了。深度学习如果离开了高性能计算的支持其能力将大打折扣，本章我们选取了 Google 的 TensorFlow 开源深度学习库作为敲门砖，帮你快速地进入高性能编程世界。那么为什么要选择 TensorFlow 呢？很简单，因为它发展实在是太快了。如图 8-1 所示，是 2017 年首届 TensorFlow 开发者大会展示的 TensorFlow 与其他深度学习平台在 GitHub 上的发展趋势统计图。TensorFlow 只花了一年的时间就已经远超了其他的深度学习平台，并且这种趋势还在不断扩大。目前 TensorFlow 1.0 版本已经可以支持 Windows 平台，这给开发者带来了很大的方便。接下来我们就正式开始我们的 TensorFlow 入门之旅。

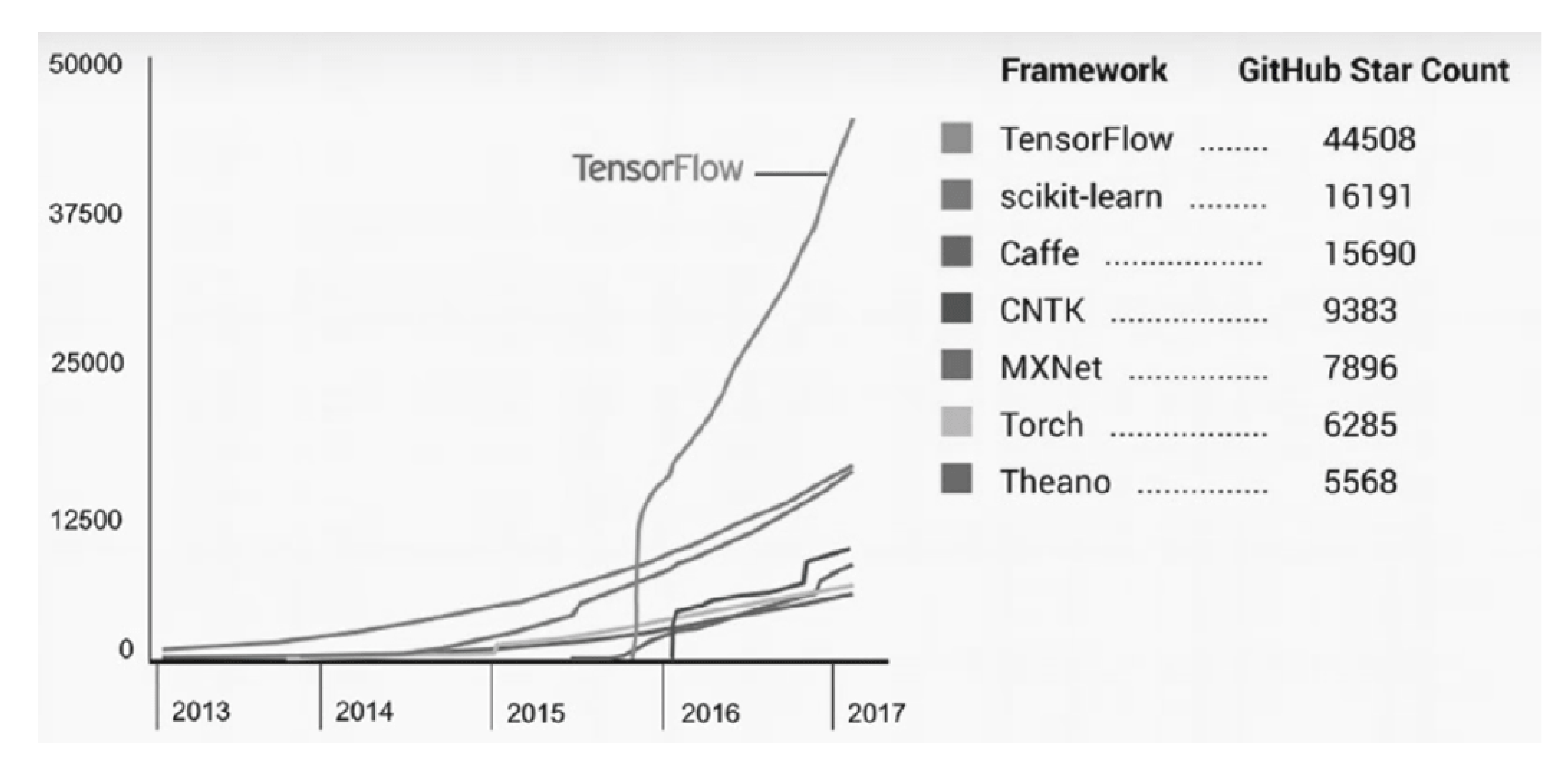


图 8-1 TensorFlow 发展趋势图

8.1 TensorFlow 介绍

TensorFlow 是一款使用**数据流图**（data flow graphs）进行数值计算的开源软件库。其中**节点**（Nodes）在数据流图中表示数学操作，**边**（Edges）则表示在节点间相互通信的多维数组或**张量**（Tensor）。该框架的结构非常灵活，只需要通过单一的 API 接口就可将 TensorFlow 部署到笔记本、服务器、移动设备中，然后使用一个或多个 CPU（或 GPU）进行计算。TensorFlow 最初是由隶属于 Google 机器智能研究机构的《Google 大脑》项目团队的研究员和工程师们所研发，并用于机器学习和深度神经网络方面的研究。相比于其他的深度学习库，TensorFlow 有着以下显著特征。

- 高度的灵活性

TensorFlow 不是一个严格的神经网络工具库，如果你能将计算模型表示为数据流图形式，就可以使用 TensorFlow 进行计算。你只需构建数据流图，描写驱动计算的内部循环，便可以利用 TensorFlow 提供的工具来帮助你组装计算子图（常用于神经网络）。用户也可以在 TensorFlow 基础上编写个性化的高层学习库，并且定义新的复合操作和写一个 Python 函数一样容易，也不需要担心性能损耗。当然你也可以自己写一点 C++代码来丰富 TensorFlow 的底层操作。

- 真正的可移植性

TensorFlow 可以运行在台式机、服务器、手机等设备中。如果你灵机一动有了机器学习的新想法，但却没有待在实验室，可以享用配置高昂的计算设备。这时，只需要轻松地打开自己的笔记本便可快速地使用 TensorFlow 实现算法。如果你想将训练模型在多个 CPU 上分布式运算又不想修改代码，TensorFlow 可以轻松地办到。如果你想要将训练好的模型作为产品的一部分应用到手机 App 里，TensorFlow 也可以轻松地办到。甚至你想要将训练好的模型作为云端服务运行在自己的服务器上，或者运行在 Docker 容器里，Tensorfow 也能快速地办到。

- 科研和产品同步

过去如果要将科研中的机器学习想法应用到产品中，需要大量的代码重写工作。但在 Google，科学家用 TensorFlow 尝试新的算法，产品团队则用 TensorFlow 来训练和使用计算模型，并直接提供给在线用户。使用 TensorFlow 可以让应用型研究者将想法迅速运用到产品中，也可以让学术型研究者更直接地彼此分享代码，从而提高科研产出率。

- 自动求微分

基于梯度的机器学习算法将受益于 TensorFlow 自动求微分的能力。在 TensorFlow 中，只需要定义预测模型的结构，并将这个结构和目标函数结合在一起，在添加数据后，TensorFlow 将自动计算相关的微分导数。计算某个变量相对于其他变量的导数仅仅是通过扩展你的图来完成的，所以你能一直清楚地看到究竟在发生什么。

- 多语言支持

TensorFlow 有一个合理的 C++使用接口，也有一个易用的 Python 使用接口来构建和执行计算流图。你可以直接编写 Python/C++程序，也可以使用 IPython 交互式界面来用 TensorFlow 尝试新想法，它可以帮你将笔记、代码、可视化等有条理地归置好。当然你也可以使用自己喜欢的语言：Go、Java、Lua、Javascript 和 R 语言来构建 TensorFlow。

- 性能最优化

假设你有一个 32 核 CPU、4 个 GPU 显卡的工作站，由于 TensorFlow 给予了线程、队列、异步操作等最佳的支持，可以让你的硬件计算潜能全部发挥出来。你可以自由地将计算图中的计算元素分配到不同设备上，TensorFlow 可以帮你管理好这些不同的副本。

8.2　TensorFlow 1.0 安装指南

本节我们将介绍如何在 Win10 平台下安装 TensorFlow1.0，并使用 NVIDIA 显卡作为 TensorFlow 计算设备。目前 Windows 平台只能使用 Python3.0+版本运行 TensorFlow，并且由于编译问题，其今后也不会支持 Python2.7+版本。因此我们将使用 TensorFlow1.0+Python3.5+Cuda8.0 的配置进行安装。如果你的计算机没有 GPU，你也可以选择安装 CPU 版本的 TensorFlow，如果只安装 CPU 版本，你可以跳过 8.2.2 及 8.2.3 节，快速进入 8.2.4 节安装 TensorFlow。

8.2.1　双版本切换 Anaconda

由于本书之前的章节中我们使用的是 Python2.7+版本进行编程练习，而现在需要使用 Python3.0+版本搭建 TensorFlow，因此我们推荐使用 Python 双版本共存的方案搭建 Python。首先，读者可以使用以下网址下载 Anaconda 对应的 Python3.0+版本：https://www.continuum.io/downloads#windows。

下载完毕后，双击.exe 文件进行安装。如图 8-2 所示，我们首先选择 Anaconda 的安装路

径，本教程选择的路径是：“D:\py3”。选择完之后单击“Next”按钮进行下一步。

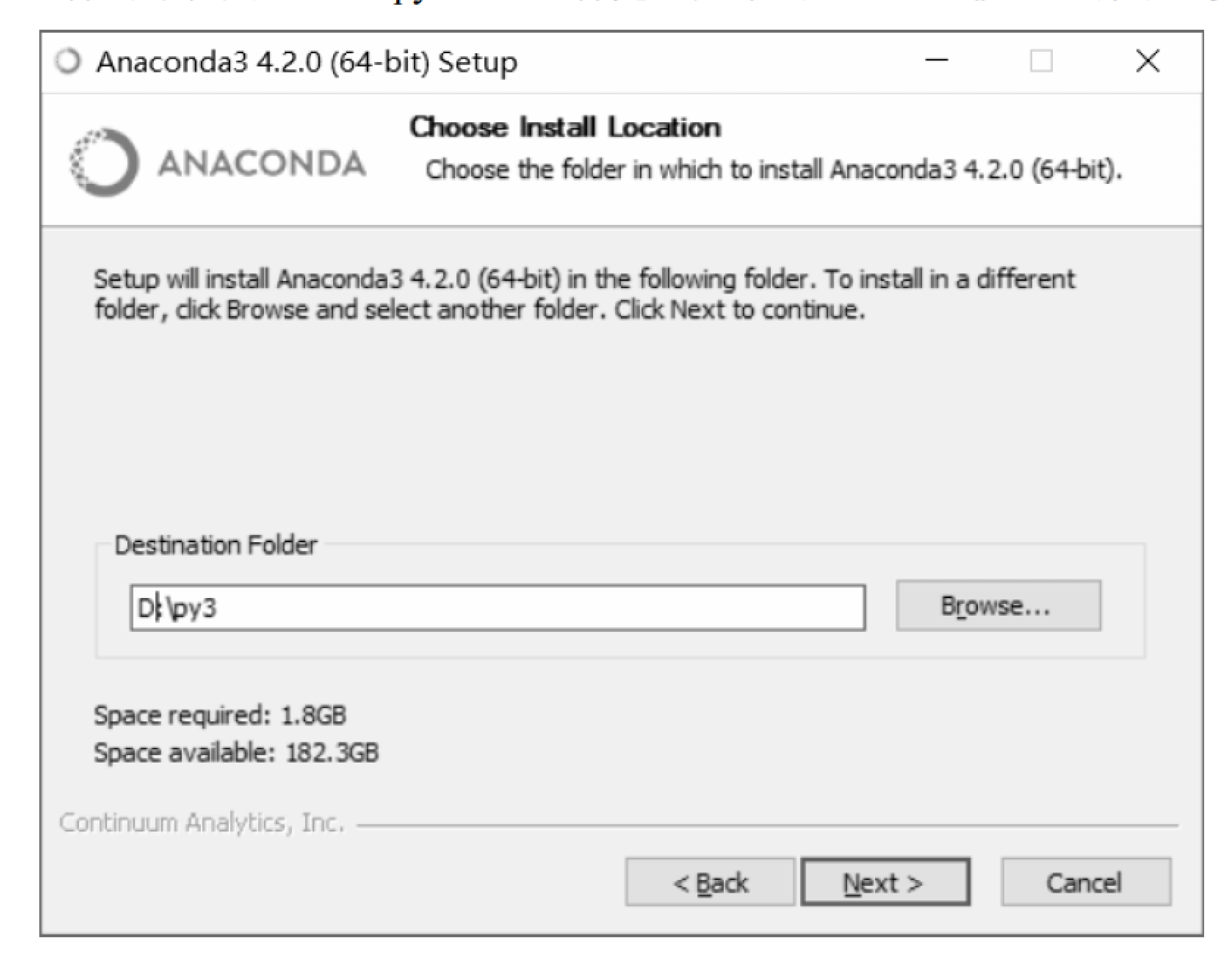


图 8-2　Anaconda 安装-路径选择

接下来安装对话框如图 8-3 所示，你需要将“Add Anaconda to my PATH environment variable”及“Register Anaconda as my default Python 3.5”两个可选项撤选，然后再单击“Install”按钮进行安装即可。

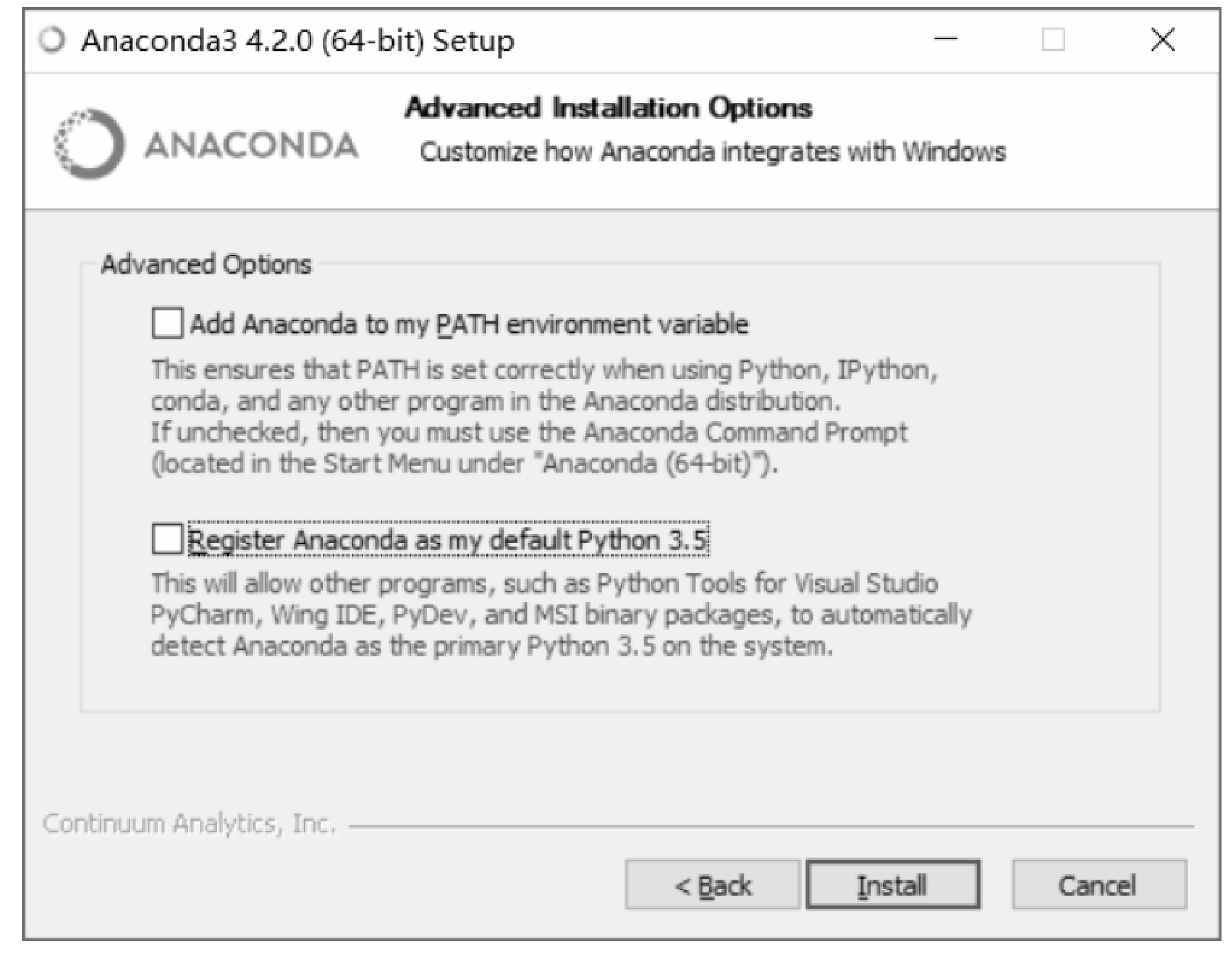


图 8-3　Anaconda 安装-高级选项选择

安装完成之后，在 CMD 里面直接输入 Python 会启动 Python2，而使用“activate d:\py3”命令之后，再使用 Python 即可切换至 Python3。如图 8-4 所示，使用“activate d:\py3”命令之后，在命令行前面会出现一个[py3]标记，此时使用任何的 Python 命令都是在 Python3 下进行的。使用 deactivate 命令可取消激活 **Python3**。

```
D:\>activate d:\py3

(d:\py3) D:\>python
Python 3.5.2 |Anaconda 4.2.0 (64-bit)| (default, Jul  5 2016, 11:41:13) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()

(d:\py3) D:\>deactivate

D:\>_
```

图 8-4　切换 Python 版本

8.2.2　安装 CUDA 8.0

CUDA（Compute Unified Device Architecture）是一种由 NVIDIA 推出的通用并行计算架构，该架构使 GPU 能够解决复杂的计算问题。它包含了 CUDA 指令集架构（ISA）以及 GPU 内部的并行计算引擎。开发人员可以使用 C 语言来为 CUDA 架构编写程序，然后在支持 CUDA 的 GPU 上进行超高性能运算。目前 CUDA 已经发展到 8.0 版本，读者可以通过以下网址下载 Windows 平台的 CUDA 8.0 的安装版本：https://developer.nvidia.com/cuda-downloads，如图 8-5 所示。



图 8-5　CUDA8.0 下载界面

CUDA 的安装也比较简单，读者只需采用安装界面的默认设置，依次单击“Next”按钮执行安装操作即可。需要注意的是，CUDA 需要 Microsoft Visual Studio 进行编译，如果没有安装过 Visual Studio，读者也可以选择在 CUDA 安装完成之后再安装 Visual Studio。

8.2.3 安装 cuDNN

cuDNN（NVIDIA CUDA® Deep Neural Network library）是一款用于深度神经网络加速的GPU 原生库。cuDNN 作为 NVIDIA 深度学习 SDK 的一部分，提供了诸如前向卷积、反向卷积、池化、归一化、激活函数等神经网络的常用函数。目前深度学习常用的框架：Caffe、TensorFlow、Theano、Torch 及 CNTK 都需要依赖于 cuDNN 的高性能 GPU 加速。

读者可以访问 https://developer.nvidia.com/cudnn 网址进行下载。该过程也非常简单，读者只需要在 NVIDIA 官网填写一些注册信息就可以选择适合你的版本进行下载。如图 8-6 所示，目前 cuDNN 有 Windows7 和 Windows10 两种版本，本书的教程使用的是 cuDNN v6.0 Library for Windows 10 版本。

cuDNN Download

NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks.

☑ I Agree To the Terms of the cuDNN Software License Agreement
Please check your framework documentation to determine the recommended version of cuDNN.
If you are using cuDNN with a Pascal (GTX 1080, GTX 1070), version 5 or later is required.

Download cuDNN v6.0 (April 27, 2017), for CUDA 8.0

Download packages updated April 27, 2017 to resolve issues related to dilated convolution on Kepler Architecture GPUs.
cuDNN User Guide
cuDNN Install Guide
cuDNN v6.0 Library for Linux
cuDNN v6.0 Library for Power8
cuDNN v6.0 Library for Windows 7
cuDNN v6.0 Library for Windows 10
cuDNN v6.0 Library for OSX
cuDNN v6.0 Release Notes
cuDNN v6.0 Runtime Library for Ubuntu16.04 (Deb)

图 8-6 cuDNN 下载页面

下载完毕后，可以将该文件解压到自定义安装路径，本书选择的路径为：“D:\cuda\”。该压缩包共有三个文件夹：bin、lib 和 include 文件，如图 8-7 所示，需要将 bin 文件夹所在路径添加到系统的环境变量中。

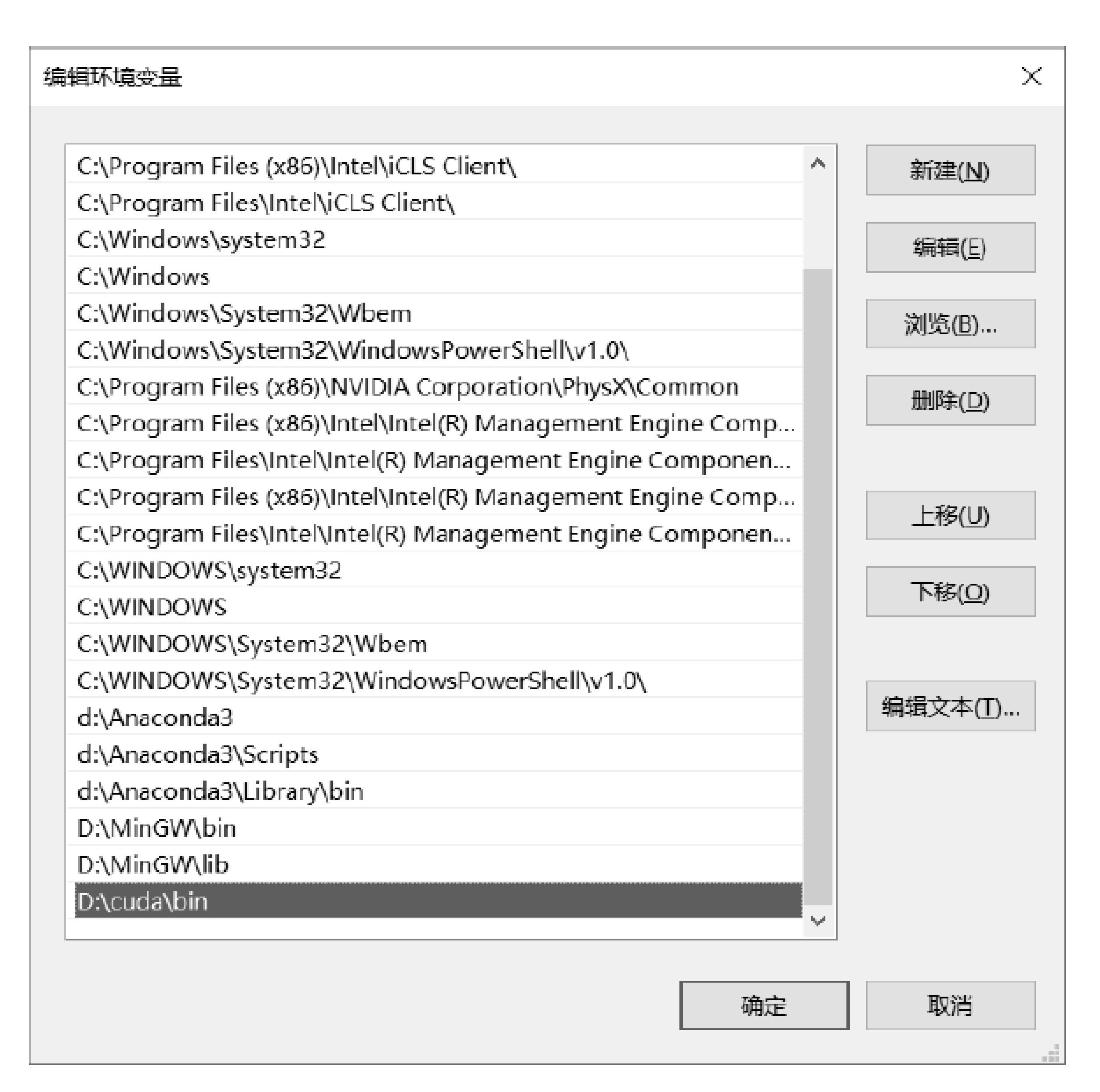


图 8-7　cuDNN 环境变量设置

8.2.4　安装 TensorFlow

安装完以上软件后，接下来我们开始安装 TensorFlow。在安装前，记得将 Anaconda 切换到 Python3.0+版本。否则系统默认会启动 Python2.7+版本的 Anaconda，这样就无法安装 TensorFlow 了。TensorFlow 的安装非常简单，如下所示，我们只需使用 pip 命令安装相应版本即可。

CPU 版本 TensorFlow：
pip install --ignore-installed --upgrade https://storage.googleapis.com/TensorFlow/windows/cpu/TensorFlow-1.1.0-cp35-cp35m-win_amd64.whl
GPU 版本 TensorFlow：
pip install --ignore-installed --upgrade https://storage.googleapis.com/TensorFlow/windows/gpu/TensorFlow_gpu-1.1.0-cp35-cp35m-win_amd64.whl

如图 8-8 所示，使用 pip 命令后，Python 会自动收集 TensorFlow 所需依赖的所有库文件，并依次下载。需要注意的是，有时因为网络拥堵原因可能会造成某些库下载失败，只需要多次使用 pip 命令安装即可。

```
命令提示符
(e:\py3) E:\>pip install --ignore-installed --upgrade https://storage.googleapis.com/tensorflow/window
s/gpu/tensorflow_gpu-1.1.0-cp35-cp35m-win_amd64.whl
Collecting tensorflow-gpu==1.1.0 from https://storage.googleapis.com/tensorflow/windows/gpu/tensorflow
_gpu-1.1.0-cp35-cp35m-win_amd64.whl
  Downloading https://storage.googleapis.com/tensorflow/windows/gpu/tensorflow_gpu-1.1.0-cp35-cp35m-wi
n_amd64.whl (48.5MB)
    100% |████████████████████████████████| 48.6MB 21kB/s
Collecting wheel>=0.26 (from tensorflow-gpu==1.1.0)
  Using cached wheel-0.29.0-py2.py3-none-any.whl
Collecting six>=1.10.0 (from tensorflow-gpu==1.1.0)
  Using cached six-1.10.0-py2.py3-none-any.whl
Collecting protobuf>=3.2.0 (from tensorflow-gpu==1.1.0)
  Using cached protobuf-3.2.0-py2.py3-none-any.whl
Collecting numpy>=1.11.0 (from tensorflow-gpu==1.1.0)
  Using cached numpy-1.12.1-cp35-none-win_amd64.whl
Collecting werkzeug>=0.11.10 (from tensorflow-gpu==1.1.0)
  Using cached Werkzeug-0.12.1-py2.py3-none-any.whl
Collecting setuptools (from protobuf>=3.2.0->tensorflow-gpu==1.1.0)
  Using cached setuptools-35.0.2-py2.py3-none-any.whl
Collecting appdirs>=1.4.0 (from setuptools->protobuf>=3.2.0->tensorflow-gpu==1.1.0)
  Using cached appdirs-1.4.3-py2.py3-none-any.whl
Collecting packaging>=16.8 (from setuptools->protobuf>=3.2.0->tensorflow-gpu==1.1.0)
  Using cached packaging-16.8-py2.py3-none-any.whl
Collecting pyparsing (from packaging>=16.8->setuptools->protobuf>=3.2.0->tensorflow-gpu==1.1.0)
  Using cached pyparsing-2.2.0-py2.py3-none-any.whl
Installing collected packages: wheel, six, appdirs, pyparsing, packaging, setuptools, protobuf, numpy,
 werkzeug, tensorflow-gpu
```

图 8-8　安装 TensorFlow

8.2.5　验证安装

安装完 TensorFlow 后，接下来我们验证 TensorFlow 的安装是否成功。你可以启动 Python 使用如下命令进行检验。

```
>>> import TensorFlow as tf
>>> hello = tf.constant( 'Hello, TensorFlow!' )
>>> sess = tf.Session( )
>>> print( sess.run( hello ) )
```

如图 8-9 所示，若命令行成功输出：b'Hello,TensorFlow!'，则表明 TensorFlow 安装成功。

```
>>> import tensorflow as tf
>>> hello = tf.constant('Hello,TensorFlow!')
>>> sess = tf.Session()
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\core\common_runt
ime\gpu\gpu_device.cc:975] Creating TensorFlow device (/gpu:0) -> (device: 0, name: GeForce
GTX 960M, pci bus id: 0000:01:00.0)
>>> print(sess.run(hello))
b'Hello,TensorFlow!'
>>> print(sess.run(hello))
b'Hello,TensorFlow!'
```

图 8-9　检验 TensorFlow 安装

- GPU 验证

如果启动 TensorFlow 时，以下 CUDA 库被成功启动，则说明 CUDA 配置成功。如果某些 CUDA 库无法启动，则应在环境变量中配置相关的路径。

```
successfully opened CUDA library cublas64_80.dll locally
successfully opened CUDA library cudnn64_5.dll locally
```

```
successfully opened CUDA library cufft64_80.dll locally
successfully opened CUDA library nvcuda.dll locally
successfully opened CUDA library curand64_80.dll locally
```

以上便是安装 TensorFlow 的整个过程，经过烦琐的安装后，接下来我们就进入 TensorFlow 的编程世界。

8.3　TensorFlow 基础

本节的主要内容截取自 TensorFlow 官方网站的 Getting Started With TensorFlow 教程。读者可以使用以下网址来查看原始的英文教程（需要使用 VPN）：

https://www.TensorFlow.org/get_started/get_started。

本教程用于指导 TensorFlow 的入门编程。在开启练习前，请先确定自己已经安装了 TensorFlow。作为本教程的前置条件，需要先了解以下内容。

- 如何使用 Python 编程;
- 至少了解少量的数组知识;
- 理想情况下应该知道机器学习。当然，如果你对机器学习了解较少，甚至毫不知情，你仍然可以使用该教材。

TensorFlow 提供了多种 API 接口。其中处于底层的 API 是 TensorFlow Core API，该接口提供了完整的编程控制，对于机器学习研究者以及需要精确控制自己模型的研发人员而言，可以使用此 API 进行科学研究。高层的 API 构建在 TensorFlow 核心接口之上，其相对核心接口而言更简单，也更容易学习使用。高层 API 使可重复性任务更容易实现，并且对于不同的用户也有更多的一致性。比如 tf.contrib.learn 可以帮助用户管理数据集、学习器、训练和执行等内容。但需要注意的是，一些高层 API 模块仍然在开发中，其命名有可能会被更迭。本教程将从 TensorFlow 核心 API 开始讲起，之后我们会使用 tf.contrib.learn 实现相同的模型。接下来打开“第 8 章-TensorFlow 初步.ipynb”文件，开始本节的练习。

8.3.1　Tensor

Tensor（张量）是 TensorFlow 的核心数据单元。一个 tensor 可以简单地理解为任意维的数组，其中 Tensor 的秩(rank)表示其维度数量。tensor 在 0 维时表示标量，也就是一个实数；一维时表示向量；二维时表示矩阵；而三维以上就表示张量。如下列代码所示。

```
3 # 秩为 0 的 tensor；其表示形状为[ ]的标量。
[1. ,2., 3.] # 秩为 1 的 tensor；其表示形状为[ 3 ]的向量。
[ [1., 2., 3.], [4., 5., 6.] ] # 秩为 2 的 tensor；其表示形状为[ 2, 3 ]的矩阵。
[ [ [ 1., 2., 3. ] ], [ [ 7., 8., 9.] ] ] # 秩为 3 的 tensor；其表示形状为[ 2, 1, 3 ]的张量。
```

8.3.2 TensorFlow 核心 API 教程

- 导入 TensorFlow

在 TensorFlow 规范的编程习惯中，使用以下语句导入 TensorFlow 库，在默认情况下我们都会使用 tf 表示 TensorFlow。

```
import TensorFlow as tf
```

- 计算图

TensorFlow 核心程序由两块单独的部分构成。

1. 构建计算图；
2. 运行计算图。

一个计算图是一系列排列好的 TensorFlow 图节点操作。每个节点使用 0 或多个 Tensor 作为输入，并且生成一个 Tensor 作为输出。在 TensorFlow 中，常数是一种特定的节点，其不需要输入，而其输出是本身内部存储的值。

接下来我们构建一个简单的计算图，如下所示，我们创建两个浮点数 node1 和 node2。

<table>
<tr><td colspan="2">node1 = tf.constant(3.0, tf.float32)
node2 = tf.constant(4.0) # 隐式地生成 tf.float32
print(node1, node2)</td></tr>
<tr><td>输出结果：</td><td>Tensor("Const:0", shape = ( ), dtype = float32)
Tensor("Const_1:0", shape = ( ), dtype = float32)</td></tr>
</table>

请注意，节点打印的结果可能不是你期望的输出值 3.0 及 4.0。因为这些节点只有在被计算时才会分别生成 3.0 以及 4.0。为了计算这些节点，我们必须通过 Session（会话）运行计算图。Session 封装了 TensorFlow 运行时的控制和状态操作，计算图只能通过 Session 运行。如下列代码所示，我们创建了一个 Session 对象，然后调用其 run 方法去运行计算图 node1 和 node2。

<table>
<tr><td colspan="2">sess = tf.Session( )
print(sess.run([node1, node2]))</td></tr>
<tr><td>输出结果：</td><td>[3.0, 4.0]</td></tr>
</table>

我们还可以通过 TensorFlow 操作（操作也是节点）来组合节点从而构建更复杂的计算图。如下列代码所示，我们可以通过将两个节点使用 add 操作来生成新的计算图。

<table>
<tr><td colspan="2">node3 = tf.add(node1, node2)
print("node3: ", node3)
print("sess.run(node3): ", sess.run(node3))</td></tr>
<tr><td>输出结果：</td><td>node3: Tensor("Add:0", shape = ( ), dtype = float32)
sess.run(node3):   7.0</td></tr>
</table>

TensorFlow 还提供了 TensorBoard 工具可视化计算图。如图 8-10 所示为上述的加法计算图可视化结果。

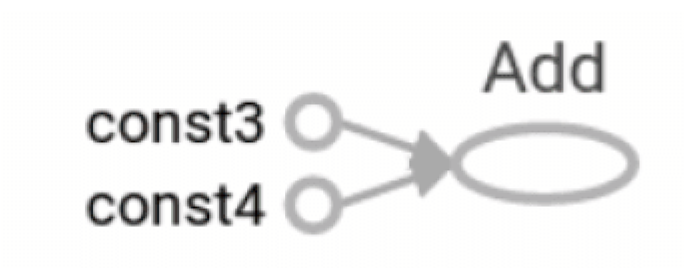


图 8-10　常数节点加法操作计算图

就目前而言，由于其只能生成常数结果，这些计算图可能还引发不了你的兴趣。为了输出可变结果，计算图还可以通过**占位符**（placeholder）进行参数化，从而接收外部输入。如下列代码所示，一个占位符也可以看作是对某类型变量的声明。

```
a = tf.placeholder( tf.float32 )
b = tf.placeholder( tf.float32 )
adder_node = a + b   # “+”是 tf.add(a, b)的简洁表达。
```

这三行代码有点类似于一个函数，我们首先定义两个输入参数（a 和 b），然后使用它们进行运算。如下列代码所示，我们可以将 Tensor 作为输入数据，给这些占位符提供具体的值进行计算。

```
print( sess.run( adder_node, { a: 3, b: 4.5 } ) )
print( sess.run( adder_node, { a: [ 1, 3 ], b: [ 2, 4 ] } ) )
```

输出结果：	7.5 [3.　7.]

在 TensorBoard 中，该计算图如图 8-11 所示。

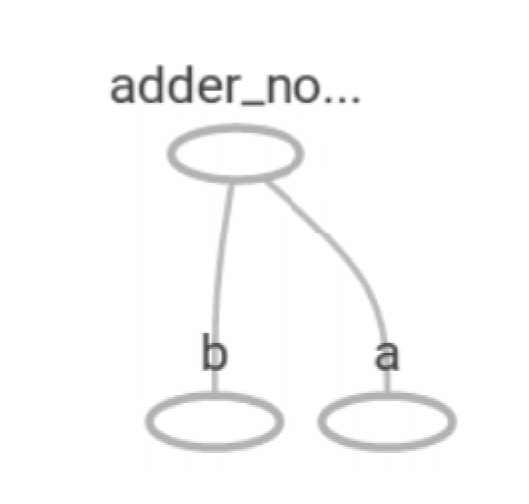


图 8-11　变量节点加法操作计算图

同样地，我们还可以在该计算图中添加额外的操作，生成更复杂的计算图。我们可以在上述的加法操作中再添加乘法操作，如下列代码所示。

```
add_and_triple = adder_node * 3.
print( sess.run( add_and_triple, { a: 3, b:4.5 } ) )
```

输出结果：	22.5

而现在的计算图就变成了图 8-12 所示。

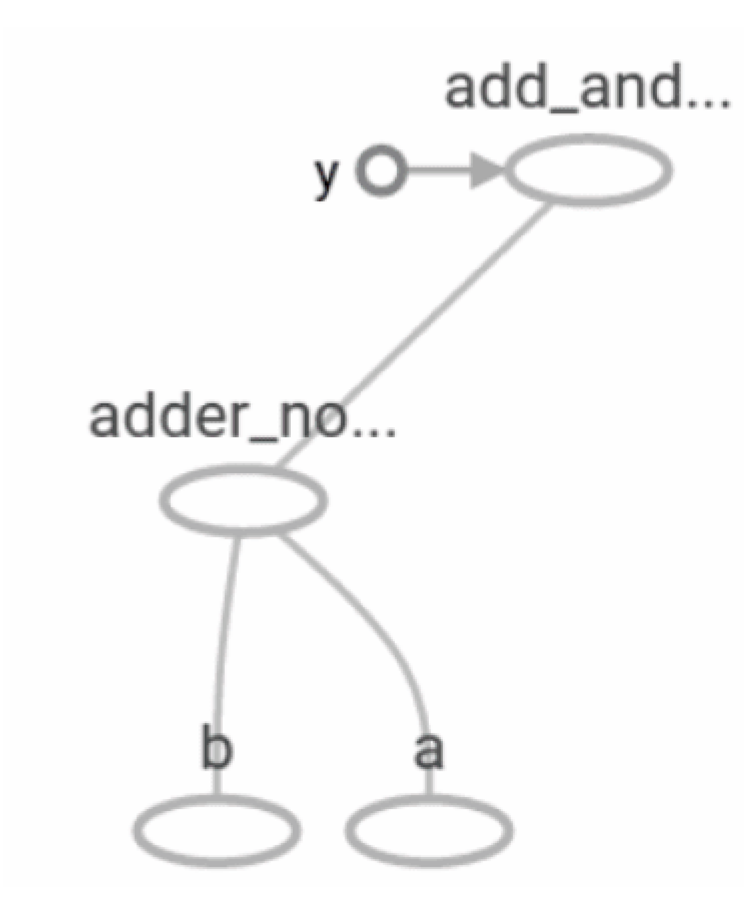


图 8-12　复合运算计算图

就如前面的内容所示，在机器学习中我们通常想要一个学习模型能够任意地获取输入数据，然后训练模型。为了满足此需求，我们就需要一个可更改的计算图，使其在相同的输入时拥有新的输出结果。而 TensorFlow 中的 Variavles（变量）就允许我们在计算图中添加可训练参数。如下列代码所示，便是在 TensorFlow 中变量的构造方式。

```
W = tf.Variable( [ .3 ], tf.float32 )
b = tf.Variable( [ -.3 ], tf.float32 )
x = tf.placeholder( tf.float32 )
linear_model = W * x + b
```

在 TensorFlow 中，常数是通过调用 tf.constant 函数进行初始化的，一旦初始化后它们的值就不会改变，但是变量在调用 tf.Variable 函数后并没有被初始化。想要在 TensorFlow 程序中初始化所有变量，如下列代码所示，必须显式地调用一个特殊的函数。

```
init = tf.global_variables_initializer( )
sess.run( init )
```

需要注意的是，init 函数用于处理 TensorFlow 子图中所有全局变量的初始化工作。在我们调用 sess.run 函数之前，变量都不会被初始化。如下列代码所示，由于 x 是一个占位符，当执行 linear_model 函数时，我们可以同时地使用一系列的 x 值作为输入。

print(sess.run(linear_model, { x:[1, 2, 3, 4] }))	
输出结果：	[0.　0.30000001　0.60000002　0.90000004]

我们已经创建了一个线性模型，但我们并不知道该模型的性能如何。想要使用训练数据训练该模型，我们需要 y 占位符作为类标，并且还需要一个损失函数。我们将使用标准的均方误差函数作为该线性回归的损失模型，其仅仅是当前模型与训练数据误差的平方再求和。如下列代码所示，linear_model - y 创建了一个向量，其每一元素对应着预测值与真实值的差值。我们通过调用 tf.square 函数对该差值进行平方运算，然后使用 tf.reduce_sum 函数将所有平方误差的值进行累加，形成一个标量损失值。

```
y = tf.placeholder( tf.float32 )
squared_deltas = tf.square( linear_model – y )
loss = tf.reduce_sum( squared_deltas )
print( sess.run( loss, { x:[ 1, 2, 3, 4 ], y:[ 0, -1, -2, -3 ] } ) )
```

输出结果：	23.66

一个变量可以在初始化时设定特定的值，也可以通过 tf.assign 函数进行赋值。例如，W=-1 以及 b=1 是本模型的最佳参数。如下列代码所示，我们可以手动地使用 tf.assign 函数将 w 和 b 的值重新修改为-1 和 1，并将损失值降到 0。

```
fixW = tf.assign( W, [ -1. ] )
fixb = tf.assign( b, [ 1. ] )
sess.run( [ fixW, fixb ] )
print( sess.run( loss, { x:[ 1, 2, 3, 4 ], y:[ 0, -1, -2, -3 ] } ) )
```

输出结果：	0.0

在本例子中，我们站在了上帝视角事先知道了 W 和 b 的最佳值，但并没有使用学习的方式自动地去寻找最佳模型。接下来我们将正式地使用一个简单的线性模型演示在 TensorFlow 中如何进行机器学习。

8.3.3　tf.train API

TensorFlow 提供了**优化器**（optimizers）去逐步地最小化损失函数。而最简单的优化器是**梯度下降**（gradient descent）优化器，如果对此方法有些陌生，该原理在本书的第 2 章可以找到。梯度下降法是通过计算损失函数的梯度来修改权重变量的值，但通常手动地计算梯度是烦琐而易错的。幸运的是，TensorFlow 的一大优势就是具备自动求导功能，该方法被封装进了 tf.gradients 中。为了简化，TensorFlow 中的优化器通常隐式地完成了这些内容。

```
optimizer = tf.train.GradientDescentOptimizer( 0.01 )
train = optimizer.minimize( loss )
sess.run( init ) # 将上述的权重变量重新初始化。
for i in range( 1000 ):
  sess.run( train, { x:[ 1, 2, 3, 4 ], y:[ 0, -1, -2, -3 ] } )
print( sess.run( [ W, b ] ) )
```

输出结果：	[array([-0.9999969], dtype = float32), array([0.99999082], dtype = float32)]

就是那么简单，只需要几行代码就完成了简单的线性回归任务。当然，对于复杂一些的模型，还需要定制输送数据到模型中的方法。因此 TensorFlow 还提供了高层抽象的 API 用于构造相同的模式、结构、功能等内容。接下来我们将学习如何使用这些高层抽象的 API。

- 完整的线性回归模型

如下列代码所示，是完整的可训练线性回归模型。

```
import numpy as np
import TensorFlow as tf
# 模型参数。
W = tf.Variable( [ .3 ], tf.float32 )
b = tf.Variable( [ -.3 ], tf.float32 )
# 模型的输入与输出。
x = tf.placeholder( tf.float32 )
linear_model = W * x + b
y = tf.placeholder( tf.float32 )
# 损失函数。
loss = tf.reduce_sum( tf.square( linear_model – y ) ) # 平方和累加。
# 优化器。
optimizer = tf.train.GradientDescentOptimizer( 0.01 )
train = optimizer.minimize( loss )
# 训练数据。
x_train = [ 1, 2, 3, 4 ]
y_train = [ 0, -1, -2, -3 ]
# 训练过程。
init = tf.global_variables_initializer( )
sess = tf.Session( )
sess.run( init ) # 初始化变量。
for i in range( 1000 ):
  sess.run( train, { x: x_train, y: y_train } )
# 计算训练精度。
curr_W, curr_b, curr_loss  = sess.run( [ W, b, loss ], { x: x_train, y: y_train } )
print( "W: %s b: %s loss: %s" % ( curr_W, curr_b, curr_loss ) )
```

输出结果：	W: [-0.9999969] b: [0.99999082] loss: 5.69997e-11

如图 8-13 所示，在 TensorBoard 中，我们可以看到完整的计算图模型。

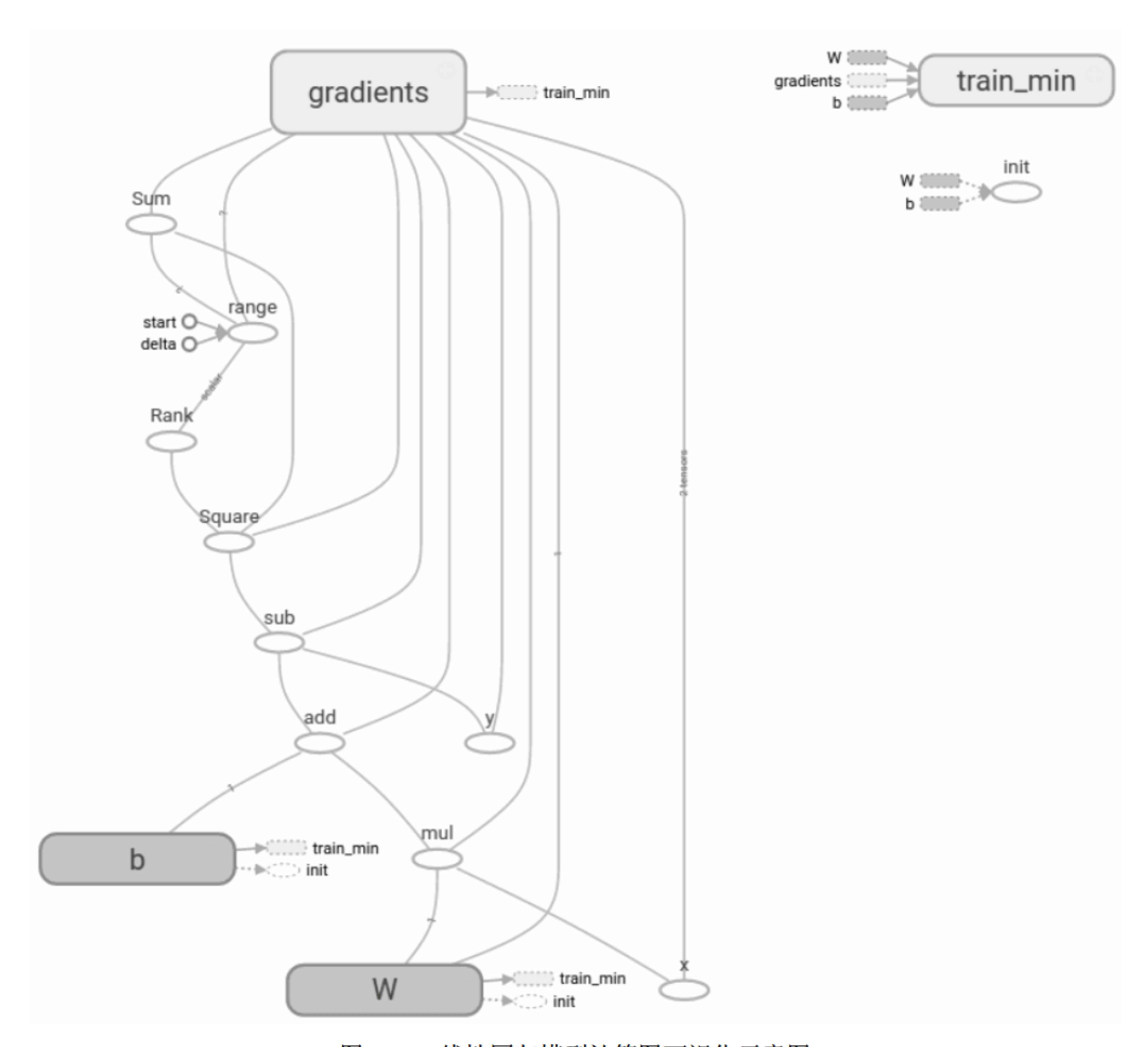


图 8-13　线性回归模型计算图可视化示意图

8.3.4　tf.contrib.learn

tf.contrib.learn 是高层的 TensorFlow 库，其将机器学习过程简化为以下几步。

- 训练阶段执行
- 评估阶段执行
- 数据集管理
- 数据供给管理

接下来，我们将展示如何使用 tf.contrib.learn 接口简化线性回归程序。

```
import TensorFlow as tf
# NumPy 会经常用于对数据进行载入、操作与处理。
import numpy as np
# 特征声明。
```

```
# TensorFlow 提供了许多复杂有用的特征类型，在本例中我们只使用一维的实值特征。
features = [ tf.contrib.layers.real_valued_column( "x", dimension = 1 ) ]
# estimator（估计器）用于训练和评估。TensorFlow 已经预定义了许多类型的估计器，
# 例如：线性回归、逻辑回归、线性分类器、逻辑分类器及大量的神经网络分类器
# 和回归器。下列代码是线性回归估计器的示例。
estimator = tf.contrib.learn.LinearRegressor( feature_columns = features )
# TensorFlow 还提供许多方法用于读取和设置数据集。
# 我们使用 numpy_input_fn 函数，并告诉函数有多少批次数据
# 进行训练( num_epochs )，以及每批数据采样多少条( batch_size )。
x = np.array( [ 1., 2., 3., 4. ] )
y = np.array( [ 0., -1., -2., -3. ] )
input_fn = tf.contrib.learn.io.numpy_input_fn( { "x": x }, y, batch_size = 4,
                                               num_epochs = 1000 )
# 我们通过 fit 方法调用 1000 次训练步传送训练数据进行训练。
estimator.fit( input_fn = input_fn, steps = 1000 )
# 我们使用 evaluate 方法评估我们训练的模型如何。在真实的例子中，
# 我们需要将数据分成验证数据及测试数据以防止过拟合。
estimator.evaluate( input_fn = input_fn )
```

输出结果：

```
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow:Saving checkpoints for 1 into
C:\Users\dufresne\AppData\Local\Temp\tmpi23p9ku4\model.ckpt.
INFO:TensorFlow:loss = 2.0, step = 1
INFO:TensorFlow:global_step/sec: 448.445
INFO:TensorFlow:loss = 0.0389375, step = 101
INFO:TensorFlow:global_step/sec: 596.601
INFO:TensorFlow:loss = 0.0082407, step = 201
INFO:TensorFlow:global_step/sec: 623.541
INFO:TensorFlow:loss = 0.000423451, step = 301
INFO:TensorFlow:global_step/sec: 636.525
INFO:TensorFlow:loss = 9.56458e-05, step = 401
INFO:TensorFlow:global_step/sec: 665.327
INFO:TensorFlow:loss = 8.59048e-06, step = 501
INFO:TensorFlow:global_step/sec: 651.423
INFO:TensorFlow:loss = 5.93324e-07, step = 601
INFO:TensorFlow:global_step/sec: 667.051
INFO:TensorFlow:loss = 1.89528e-07, step = 701
INFO:TensorFlow:global_step/sec: 664.496
INFO:TensorFlow:loss = 1.45992e-08, step = 801
```

```
INFO:TensorFlow:global_step/sec: 694.997
INFO:TensorFlow:loss = 1.65755e-10, step = 901
INFO:TensorFlow:Saving checkpoints for 1000 into
C:\Users\dufresne\AppData\Local\Temp\tmpi23p9ku4\model.ckpt.
INFO:TensorFlow:Loss for final step: 4.13461e-10.
INFO:TensorFlow:Starting evaluation at 2017-05-04-03:16:40
INFO:TensorFlow:Finished evaluation at 2017-05-04-03:16:41
INFO:TensorFlow:Saving dict for global step 1000: global_step = 1000, loss = 1.77339e-10
WARNING:TensorFlow:Skipping summary for global_step, must be a float or np.float32.
{'global_step': 1000, 'loss': 1.7733889e-10}
```

- 自定义模型

tf.contrib.learn 除了拥有预定义的模型外，还可以自定义模型。假设我们想要创建一个 TensorFlow 还没实现的自定义模型，可能仍然需要保留 tf.contrib.learn 中诸如数据集、数据供给、训练等抽象方法。接下来，我们将展示如何通过低层的 TensorFlow 接口实现自定义的线性回归器 LinearRegressor。

想要通过 tf.contrib.learn 接口自定义学习模型，我们需要使用到 tf.contrib.learn.Estimator 类作为自定义类的父类，因此 tf.contrib.learn.LinearRegressor 需要继承于 tf.contrib.learn.Estimator 类。但和 Estimator 的其他子类不同，我们只是简单地提供 model_fn 给 Estimator，并告诉 tf.contrib.learn 评估预测、训练次数、损失等方法即可。

```
import numpy as np
import TensorFlow as tf
def model( features, labels, mode ) :
# 构建线性模型。
W = tf.get_variable( "W", [ 1 ], dtype = tf.float64 )
b = tf.get_variable( "b", [ 1 ], dtype = tf.float64 )
y = W * features[ 'x' ] + b
# 损失子图。
loss = tf.reduce_sum( tf.square( y – labels ) )
# 训练子图。
global_step = tf.train.get_global_step( )
optimizer = tf.train.GradientDescentOptimizer( 0.01 )
train = tf.group(optimizer.minimize( loss ), tf.assign_add( global_step, 1 ) )
# ModelFnOps 用于连接我们构建的方法子图。
return tf.contrib.learn.ModelFnOps( mode = mode, predictions = y,
loss = loss, train_op = train )
estimator = tf.contrib.learn.Estimator( model_fn = model )
# 定义数据集。
x = np.array( [ 1., 2., 3., 4. ] )
```

```
y = np.array( [ 0., -1., -2., -3. ] )
input_fn = tf.contrib.learn.io.numpy_input_fn( { "x": x }, y, 4, num_epochs = 1000 )
# 训练模型。
estimator.fit( input_fn = input_fn, steps = 1000 )
# 评估模型。
print( estimator.evaluate( input_fn = input_fn, steps = 10 ) )
```

输出结果：

```
WARNING:TensorFlow:Using temporary folder as model directory:
C:\Users\dufresne\AppData\Local\Temp\tmppgx3im5k
INFO:TensorFlow:Using default config.
INFO:TensorFlow:Using config: {'_save_summary_steps': 100, '_save_checkpoints_steps': None,
'_keep_checkpoint_every_n_hours': 10000, '_keep_checkpoint_max': 5, '_tf_config': gpu_options {
per_process_gpu_memory_fraction: 1
}
, '_cluster_spec': <TensorFlow.python.training.server_lib.ClusterSpec object at 0x0000029060DE65F8>,
'_evaluation_master': '', '_tf_random_seed': None, '_is_chief': True, '_task_id': 0, '_save_checkpoints_secs': 600,
'_num_ps_replicas': 0, '_master': '', '_task_type': None, '_environment': 'local'}
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow:Saving checkpoints for 1 into
C:\Users\dufresne\AppData\Local\Temp\tmppgx3im5k\model.ckpt.
INFO:TensorFlow:loss = 5.20388559268, step = 1
INFO:TensorFlow:global_step/sec: 834.604
INFO:TensorFlow:loss = 1.58416664772, step = 101
INFO:TensorFlow:global_step/sec: 906.577
INFO:TensorFlow:loss = 0.0310900932304, step = 201
INFO:TensorFlow:global_step/sec: 1041.88
INFO:TensorFlow:loss = 0.00538012149811, step = 301
INFO:TensorFlow:global_step/sec: 991.046
INFO:TensorFlow:loss = 6.32636899449e-08, step = 401
INFO:TensorFlow:global_step/sec: 1089.81
INFO:TensorFlow:loss = 3.69894665652e-05, step = 501
INFO:TensorFlow:global_step/sec: 1060.97
INFO:TensorFlow:loss = 3.17184819319e-06, step = 601
INFO:TensorFlow:global_step/sec: 1016.39
INFO:TensorFlow:loss = 2.54515951329e-07, step = 701
INFO:TensorFlow:global_step/sec: 1101.29
INFO:TensorFlow:loss = 1.63026604358e-08, step = 801
INFO:TensorFlow:global_step/sec: 1016.04
INFO:TensorFlow:loss = 2.20882298833e-09, step = 901
```

```
INFO:TensorFlow:Saving checkpoints for 1000 into
C:\Users\dufresne\AppData\Local\Temp\tmppgx3im5k\model.ckpt.
INFO:TensorFlow:Loss for final step: 1.49740992028e-10.
INFO:TensorFlow:Starting evaluation at 2017-05-04-06:46:17
INFO:TensorFlow:Evaluation [1/10]
INFO:TensorFlow:Evaluation [2/10]
INFO:TensorFlow:Evaluation [3/10]
INFO:TensorFlow:Evaluation [4/10]
INFO:TensorFlow:Evaluation [5/10]
INFO:TensorFlow:Evaluation [6/10]
INFO:TensorFlow:Evaluation [7/10]
INFO:TensorFlow:Evaluation [8/10]
INFO:TensorFlow:Evaluation [9/10]
INFO:TensorFlow:Evaluation [10/10]
INFO:TensorFlow:Finished evaluation at 2017-05-04-06:46:17
INFO:TensorFlow:Saving dict for global step 1000: global_step = 1000, loss = 2.21911e-10
WARNING:TensorFlow:Skipping summary for global_step, must be a float or np.float32.
{'loss': 2.2191052e-10, 'global_step': 1000}
```

8.4　TensorFlow 构造 CNN

在本小节的编程练习中，我们将使用 TensorFlow 构建 Softmax 分类器以及卷积神经网络。接下来，你可以使用“第 8 章-TensorFlow 构造 CNN 初步.ipynb”文件完成本节练习。本节中我们将逐步完成以下操作。

1. 如何使用 TensorFlow 创建 Softmax 分类器识别 MNIST 数字图形数据集；
2. 如何使用 TensorFlow 训练模型；
3. 如何使用 TensorFlow 测试模型精度；
4. 如何使用 TensorFlow 创建卷积神经网络并训练模型。

8.4.1　构建 Softmax 模型

- 载入 MNIST 数据集

如下列代码所示，我们可以使用 TensorFlow 自动地下载和读取 MNIST 数据集。

```
from TensorFlow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets( 'MNIST_data', one_hot = True )
```

输出结果：	Extracting MNIST_data\train-images-idx3-ubyte.gz Extracting MNIST_data\train-labels-idx1-ubyte.gz

	Extracting MNIST_data\t10k-images-idx3-ubyte.gz Extracting MNIST_data\t10k-labels-idx1-ubyte.gz

MNIST 是一个使用 NumPy 数组存储训练、验证、测试数据的轻量级类，它还提供了批量采样数据等方法，将会在接下来的内容中使用。

- 启动 TensorFlow InteractiveSession

TensorFlow 使用高效的 C++代码在后端进行计算，而 Session 便是我们连接到后端的桥梁。在 TensorFlow 中，通常首先会创建计算图，然后再使用 Session 启动计算图。但和上小节内容不同，接下来我们使用 InteractiveSession 类创建 Session，如下列代码所示，该类可以帮你在 TensorFlow 中更灵活地构建代码。如果不使用 InteractiveSession，那需要在运行计算图前构建好完整的计算图。InteractiveSession 允许你交错地构建和运行计算图，这在使用诸如 IPython 这样的交互式文本中显得非常的方便。

```
import TensorFlow as tf
sess = tf.InteractiveSession( )
```

- 计算图

为了能够在 Python 中高效地进行数值计算，我们通常会使用 NumPy 库在 Python 的外部使用别的高效率语言，执行类似矩阵乘法这样的高负荷操作。但不幸的是，在每一次 Python 的前后端切换操作时都需要大量的额外开销。特别是使用 GPU 或者分布式策略时，这种数据转换开销将异常耗时。TensorFlow 同样采用在 Python 外部执行的方式提高效率，但其也通过其他一些措施来避免这种额外的开销。和在 Python 外运行单独的高负荷操作不同，TensorFlow 允许我们描述交互操作图，然后将所有操作完全运行于 Python 之外。这种方式和 Theano、Torch 等主流深度学习库相似。因此在 TensorFlow 中，Python 代码只是负责构建外部的计算图，然后控制计算图运行。

在本小节中，我们将通过一层线性层构建 Softmax 模型。然后在下一小节中，我们将在此基础上构建卷积神经网络。

- 占位符

如下列代码所示，我们从创建用于输入图像与输出类标的计算图节点开始。

```
x = tf.placeholder( tf.float32, shape = [ None, 784 ] )
y_ = tf.placeholder( tf.float32, shape = [ None, 10 ] )
```

x 和 y_都没有特定的值，因此我们使用 **placeholder**（占位符）进行声明。当 TensorFlow 运行计算图时，我们再动态地输入数据。输入图像 x 是一个二维的浮点数 tensor，我们将其形状声明为[None,784]。其中 784 是一张 28×28 像素的 MNIST 图像的数据维度，None 指的是 Tensor 的第一维，也就是对应的图像批量尺寸可以是任意值。输出分类 y_同样也是一个二维 Tensor，其每一行是 10 维的输出向量，表示 MNIST 图像对应的 0-9 数字类标。

- 变量

接下来，我们使用 Variable（变量）定义模型的权重 W 与偏置项 b，如下列代码所示。一个变量是驻留在 TensorFlow 计算图中的值，其可以在计算时被修改。在机器学习的语境中，我们也把变量称为参数。

```
W = tf.Variable( tf.zeros( [ 784, 10 ] ) )
b = tf.Variable( tf.zeros( [ 10 ] ) )
```

我们通过调用 tf.Variable 函数将初始值传递到每一个参数中。在该例子中，我们将 W 和 b 都初始化为 0，其中 W 是 784×10 的矩阵，而 b 是一个 10 维向量。变量在被使用之前必须使用 Session 进行初始化，这一步相当于将上述初始化的值(tf.zeros)传递到变量中。

如下列代码所示，我们可以使用 tf.global_variables_initializer 函数一次性地将变量全部初始化。

```
sess.run( tf.global_variables_initializer( ) )
```

- 预测分类与损失函数

接下来我们开始 Softmax 的编码。如下列代码所示，我们只是用输入图像 x 与权重矩阵 W 进行乘法运算，然后加上偏置项 b 即可。

```
y = tf.matmul( x, W ) + b
```

如下列代码所示，我们使用交叉熵作为损失函数，该函数已经在 TensorFlow 中实现了，我们只需要传入计算图 y 与类标节点 y_即可。

```
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits( labels = y_, logits = y ) )
```

需要注意的是，tf.nn.softmax_cross_entropy_with_logits 内部使用的是非归一化（unnormalized）Softmax 模型进行预测并累加所有分类的得分。

8.4.2　使用 TensorFlow 训练模型

现在我们已经定义好了训练模型以及损失函数，接下来我们就使用 TensorFlow 进行训练。由于 TensorFlow 已经知道了完整的计算图，其可以依据损失值自动地求导，然后计算梯度修改权重变量。TensorFlow 内置了许多的优化算法，如下列代码所示，我们将学习率设置为 0.5，使用最速梯度下降法降低交叉熵代价损失。

```
train_step = tf.train.GradientDescentOptimizer( 0.5 ).minimize( cross_entropy )
```

在上述的一行代码中，TensorFlow 的工作其实是在计算图中加入新的操作节点。这些操作包括了计算梯度、计算参数更新步数与更新参数等操作。其返回值 train_step 也是一个操作，当运行计算图时，其将使用梯度更新参数。因此如下列代码所示，训练模型可以重复地运行

train_step 来不断地更新参数。

```
for _ in range( 1000 ):
    batch = mnist.train.next_batch( 100 )
    train_step.run( feed_dict = { x: batch[ 0 ], y_: batch[ 1 ] } )
```

每次迭代时，我们载入 100 条训练数据进行训练。当我们运行 train_step 操作时，使用 feed_dict 将占位符 x 与 y_替换为载入的训练数据。需要注意的是，可以在计算图中使用 feed_dict 替换任何 Tensor，不仅仅局限于占位符。

8.4.3 使用 TensorFlow 评估模型

在评估模型时，我们首先要预测类标。tf.argmax 是一个非常有用的函数，其返回给定 Tensor 某一坐标轴上最高得分的索引值。例如，tf.argmax(y,1)返回的是模型每一输入数据最大可能的预测类标，而 tf.argmax(y_,1)返回的是真实的类标。最后我们使用 tf.equal 函数检查预测类标与真实类标的一致性，如下列代码所示。

```
correct_prediction = tf.equal( tf.argmax( y, 1 ), tf.argmax( y_,1 ) )
```

该返回值 correct_prediction 为一个布尔值链表。想要计算模型的精度，我们还需要计算该链表的均值。例如，[True,False,True,True]可以用[1,0,1,1]表示，其精度为 0.75。

```
accuracy = tf.reduce_mean( tf.cast( correct_prediction, tf.float32 ) )
```

最后，我们使用测试数据评估我们模型的精确度，该测试结果大约在 92%，如下列代码所示。

```
print( accuracy.eval( feed_dict = { x: mnist.test.images, y_: mnist.test.labels } ) )
```

输出结果：	0.918

8.4.4 使用 TensorFlow 构建卷积神经网络

虽然我们的 Softmax 分类器可以在 MNIST 数据集上实现 92%的精确度，但这性能其实是比较差的。在本小节中，我们将使用 TensorFlow 构建小型的卷积神经网络，其模型精度大约在 99.2%。

- *权重初始化*

首先，我们需要创建大量的权重和偏置项参数。如下列代码所示，我们使用标准差为 0.1 的正态分布初始化权重。由于我们使用 ReLU 作为激活函数，为了避免神经元死亡现象，我们使用常数 0.1 初始化偏置项。

```
def weight_variable( shape ):
    initial = tf.truncated_normal( shape, stddev = 0.1 )
```

```
    return tf.Variable( initial )
def bias_variable( shape ):
    initial = tf.constant( 0.1, shape = shape )
    return tf.Variable( initial )
```

- 卷积和池化

TensorFlow 的卷积和池化操作具有很大的灵活性，我们可以自定义卷积核尺寸、跨步尺寸和零填充类型等功能。如下列代码所示，在卷积操作中我们使用为 1 的跨步和 same 零填充进行卷积特征提取；在池化操作时，我们使用 2×2 的最大池化操作。

```
def conv2d( x, W ):
    return tf.nn.conv2d( x, W, strides = [ 1, 1, 1, 1 ], padding = 'SAME' )
def max_pool_2x2( x ):
    return tf.nn.max_pool( x, ksize = [ 1, 2, 2, 1 ],
                                        strides = [ 1, 2, 2, 1 ], padding = 'SAME' )
```

- 构造第一层卷积层

接下来，我们实现第一层卷积层。我们使用 32 个 5×5 的卷积核作为第一层卷积特征提取层，因此第一层权重是形状为[5,5,1,32]的张量，该张量的前两维表示卷积核的大小，第三维表示输入通道，第 4 维表示输出通道。而卷积层的偏置项为 32 维的向量，对应于每一输出通道。

```
W_conv1 = weight_variable( [ 5, 5, 1, 32 ] )
b_conv1 = bias_variable( [ 32 ] )
```

为了使输入数据与卷积权重相对应，我们首先要将输入数据 x 重塑为一个四维张量。该张量的第二维与第三维分别对应图像的宽和高，最后一维对应图像的色彩通道，第一维为图像的数量。

```
x_image = tf.reshape( x, [ -1, 28, 28, 1 ] )
```

接下来我们就开始使用权重张量 W_conv1 与输入数据 x_image 进行卷积，然后加上偏置项使用 ReLU 函数进行激活，最后再进行最大池化。如下列代码所示，max_pool_2x2 函数将图片尺寸减少到 14×14。

```
h_conv1 = tf.nn.relu( conv2d( x_image, W_conv1 ) + b_conv1 )
h_pool1 = max_pool_2x2( h_conv1 )
```

- 第二层卷积层

如下列代码所示，在第二层的卷积层中，我们使用 64 个 5×5 的卷积核进行特征提取，然后再使用 2×2 的最大池化。

```
W_conv2 = weight_variable( [ 5, 5, 32, 64 ] )
b_conv2 = bias_variable( [ 64 ] )
h_conv2 = tf.nn.relu( conv2d( h_pool1, W_conv2 ) + b_conv2 )
h_pool2 = max_pool_2x2( h_conv2 )
```

- 全连接层

现在，我们已经将图片的尺寸降低到了 7×7，接下来我们添加一层 1024 单元的全连接层进行特征提取。如下列代码所示，全连接层的权重为 3136×1024，其中 3136 为第二层卷积层的输出维度，1024 为全连接层的输出维度。需要注意的是，当卷积层与全连接层进行对接时，需要将第二层卷积层输出的三维(宽，高，色道)的特征重塑为一维。

```
W_fc1 = weight_variable( [ 7 * 7 * 64, 1024 ] )
b_fc1 = bias_variable( [ 1024 ] )
h_pool2_flat = tf.reshape( h_pool2, [ -1, 7 * 7 * 64 ] )
h_fc1 = tf.nn.relu( tf.matmul( h_pool2_flat, W_fc1 ) + b_fc1 )
```

- Dropout

为了降低过拟合现象，我们在输出层之前再加入一层 Dropout 层。如下列代码所示，我们使用占位符表示 Dropout 的神经元激活概率，这允许我们可以在训练阶段开启 Dropout，而在测试阶段关闭 Dropout 功能。TensorFlow 的 tf.nn.dropout 函数已经实现了 Dropout 操作，我们直接将其当作节点添加到计算图中即可。

```
keep_prob = tf.placeholder( tf.float32 )
h_fc1_drop = tf.nn.dropout( h_fc1, keep_prob )
```

- 输出层

如下列代码所示，卷积网络的输出层和 Softmax 一样，你也可以认为我们是在全连接层之后添加了一层 Softmax 层。

```
W_fc2 = weight_variable( [ 1024, 10 ] )
b_fc2 = bias_variable( [ 10 ] )
y_conv = tf.matmul( h_fc1_drop, W_fc2 ) + b_fc2
```

- 训练和评估卷积网络

卷积网络的训练和评估过程与上述的 Softmax 基本相同，区别有以下几点。

1. 我们将最速梯度下降优化器替换成了更复杂一些的 Adam 优化器；
2. 我们在 feed_dict 中添加了额外的 keep_prob 参数控制 Dropout 激活概率；
3. 在训练过程中，每迭代 500 次我们记录一次训练情况。

```
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits( labels = y_, logits = y_conv ) )
```

```
train_step = tf.train.AdamOptimizer( 1e-4 ).minimize( cross_entropy )
correct_prediction = tf.equal( tf.argmax( y_conv, 1 ), tf.argmax( y_, 1 ) )
accuracy = tf.reduce_mean( tf.cast( correct_prediction, tf.float32 ) )
sess.run( tf.global_variables_initializer( ) )
for i in range( 10000 ):
  batch = mnist.train.next_batch( 50 )
  if i%500 == 0:
    train_accuracy = accuracy.eval( feed_dict = {
        x: batch[ 0 ], y_: batch[ 1 ], keep_prob: 1.0 } )
    print("步数 %d，训练精确度： %g"%( i, train_accuracy ) )
  train_step.run( feed_dict = { x: batch[ 0 ], y_: batch[ 1 ], keep_prob: 0.5 } )
print( "测试精确度： %g"% accuracy.eval( feed_dict = {
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0 } ) )
```

输出结果：

```
步数 0，训练精确度： 0.02
步数 500，训练精确度： 0.94
步数 1000，训练精确度： 0.96
步数 1500，训练精确度： 0.96
步数 2000，训练精确度： 1
步数 2500，训练精确度： 0.96
步数 3000，训练精确度： 1
步数 3500，训练精确度： 1
步数 4000，训练精确度： 0.98
步数 4500，训练精确度： 0.98
步数 5000，训练精确度： 1
步数 5500，训练精确度： 1
步数 6000，训练精确度： 1
步数 6500，训练精确度： 1
步数 7000，训练精确度： 1
步数 7500，训练精确度： 1
步数 8000，训练精确度： 0.98
步数 8500，训练精确度： 0.98
步数 9000，训练精确度： 0.98
步数 9500，训练精确度： 1
测试精确度： 0.9922
```

8.5　TensorBoard 快速入门

使用 TensorFlow 训练大规模深度神经网络常常就像是使用黑魔法一样，我们只知道最终

的结果，但却很难理解内部发生了什么。为了更加轻松地理解、调试与优化 TensorFlow 程序，TensorFlow 中加入了一套可视化工具——TensorBoard。

TensorBoard 可以帮助你可视化 TensorFlow 计算图，定量地绘制执行计算图时各种数据的变化情况。如图 8-14 所示，为配置好 TensorBoard 后的交叉熵损失变化曲线图。

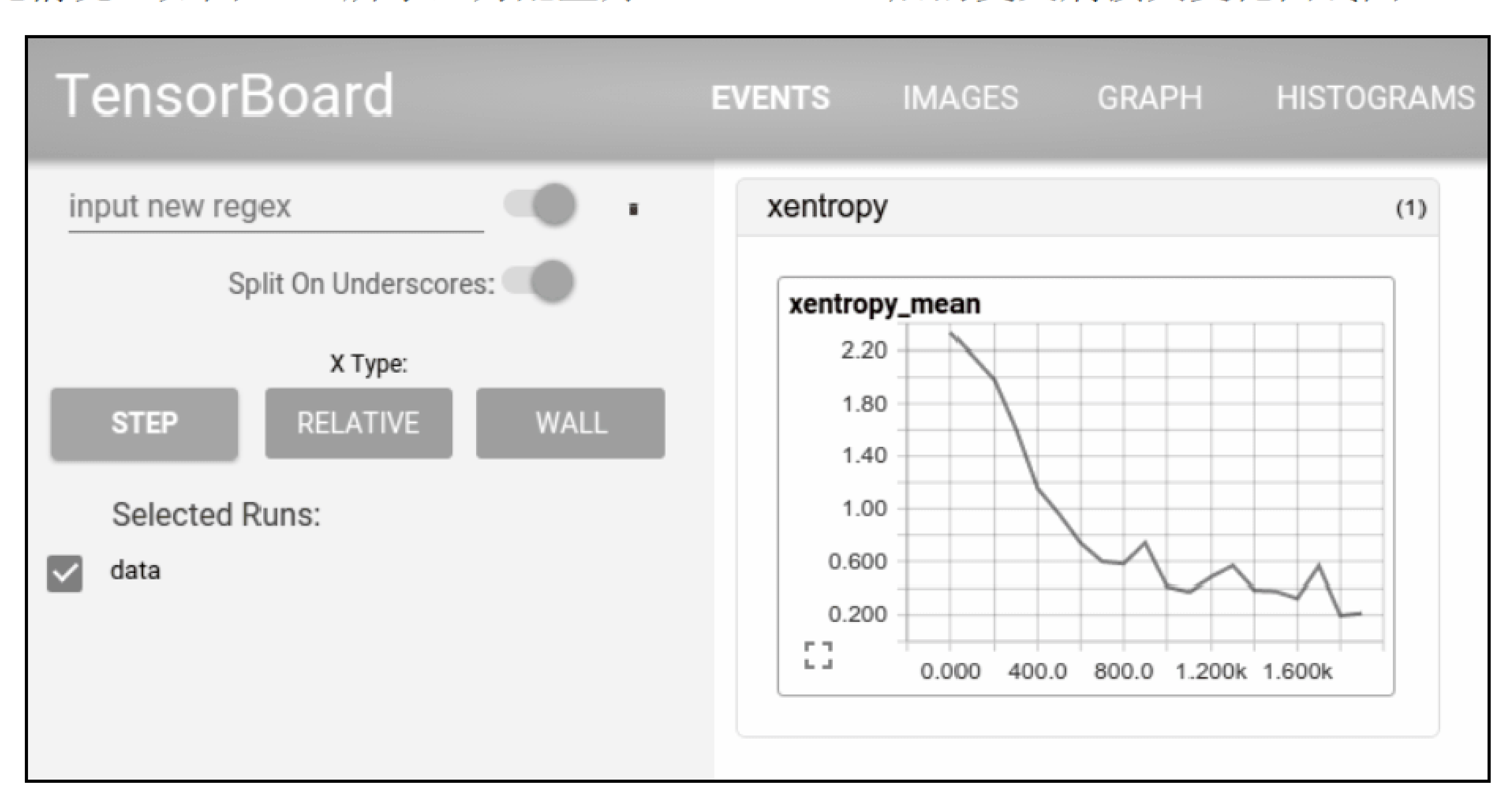


图 8-14 TensorBoard 示意图

在本小节中，我们将简单地介绍 TensorBoard 的用法。更多的资源可以访问如下网址：https://www.TensorFlow.org/code/TensorFlow/tensorboard/README.md（需要使用 VPN），来查看更多的 TensorBoard 用法、技巧与调试等信息。

8.5.1 TensorBoard 可视化学习

- *序列化数据*

当运行 TensorFlow 时，可能需要在运行过程中动态地生成一些**汇总数据**（Summary data），而 TensorBoard 就通过读取这些写入磁盘的 TensorFlow 事件文件进行可视化，以下是这些汇总数据在 TensorBoard 中的生命周期。

首先，在创建 TensorFlow 计算图时，可以通过 Summary 操作进行注释，你需要指定在哪些节点中收集汇总数据。例如，假设你训练一个卷积神经网络识别 MNIST 数字，你可能想要记录学习率及损失函数如何改变。那你就可以通过在相应的节点中附上 tf.summary.scalar 操作去收集这些信息。然后，你再给这些 scalar_summary 取个类似“学习率”“损失函数”这样有意义的标签即可。或许你也想要可视化每一层激活函数、梯度、权重的分布情况，那你可以通过在输出梯度与权重变量中附上 tf.summary.histogram 操作分别地收集这些信息。

TensorFlow 操作在你运行它们之前都不会做任何事情，但 Summary 节点对于计算图而言只是个外部设备：你当前运行的节点都不依赖于它们。因此，想要生成这些信息，我们需要先运行这些 Summary 节点。但手动管理这些节点是非常烦琐的，因此我们可以使用 tf.summary.merge_all 函数将这些节点合并在一个操作中生成所有的汇总数据，然后就只需运行合并的 Summary 操作，其会将所有的汇总数据生成序列化的 Summary protobuf 对象。最后

我们将 Summary protobuf 对象传递到 tf.summary.FileWriter 就可以将汇总数据写入到磁盘中。

FileWriter 在其构造器中需要指定一个 logdir（路径），该路径是非常重要的，它是所有写出事件的目录。FileWriter 的构造器中也能可选地传入一个计算图，如果其接收到一个计算图对象，TensorBoard 将根据 Tensor 形状信息可视化你的计算图。需要注意的是，如果需要，可以在运行每一步计算图时都记录一次日志，但这可能会有成千上万的训练信息，这可能已经远远多于你需要的跟踪信息了。因此，最好每隔 *n* 步，运行一次合并 Summary 操作。

我们将使用 TensorFlow 源码自带的教程例子来介绍 TensorBoard 的使用方法，该文件存放在 TensorFlow 源码中的“TensorFlow\examples\tutorials\mnist\ mnist_with_summaries.py”文件下，以下是配置 TensorBoard 的关键代码。

```
def variable_summaries( var ):
  """在 Tensor 中附上用于 TensorBoard 可视化的 summaries"""
  with tf.name_scope( 'summaries' ):
    mean = tf.reduce_mean( var )
    tf.summary.scalar( 'mean', mean )
    with tf.name_scope( 'stddev' ):
      stddev = tf.sqrt( tf.reduce_mean( tf.square( var – mean ) ) )
    tf.summary.scalar( 'stddev', stddev )
    tf.summary.scalar( 'max', tf.reduce_max( var ) )
    tf.summary.scalar( 'min', tf.reduce_min( var ) )
    tf.summary.histogram( 'histogram', var )
def nn_layer( input_tensor, input_dim, output_dim, layer_name, act = tf.nn.relu ):
  """ 构建简单神经网络的可复用性代码。
完成矩阵乘法，加偏置项，使用 ReLU 非线性激活等功能。
并且设置合理的命名空间，使得更容易地阅读计算图及添加 summary 操作。"""
  # 在计算图的一层中添加命名作用域。
  with tf.name_scope( layer_name ):
    # 该变量将保留这层的权重状态。
    with tf.name_scope( 'weights' ):
      weights = weight_variable( [ input_dim, output_dim ] )
      variable_summaries( weights )
    with tf.name_scope( 'biases' ):
      biases = bias_variable( [ output_dim ] )
      variable_summaries( biases )
    with tf.name_scope( 'Wx_plus_b' ):
      preactivate = tf.matmul( input_tensor, weights ) + biases
      tf.summary.histogram( 'pre_activations', preactivate )
    activations = act( preactivate, name = 'activation' )
    tf.summary.histogram( 'activations', activations )
    return activations
```

```
hidden1 = nn_layer( x, 784, 500, 'layer1' )
with tf.name_scope( 'dropout' ):
  keep_prob = tf.placeholder( tf.float32 )
  tf.summary.scalar( 'dropout_keep_probability', keep_prob )
  dropped = tf.nn.dropout( hidden1, keep_prob )
# 如下列代码所示，没有使用 Softmax 激活。
y = nn_layer( dropped, 500, 10, 'layer2', act = tf.identity )
with tf.name_scope( 'cross_entropy' ):
  # 原始的交叉熵表达式：
  # tf.reduce_mean( -tf.reduce_sum( y_ * tf.log( tf.softmax( y ) ),
  #                                 reduction_indices = [ 1 ] ) )
  # 可能造成数值不稳的现象。因此我们在原来的神经网络特征提取后，
  # 使用 tf.nn.softmax_cross_entropy_with_logits 函数，然后再求该批次数据损失均值。
  diff = tf.nn.softmax_cross_entropy_with_logits (targets = y_, logits = y )
  with tf.name_scope( 'total' ):
    cross_entropy = tf.reduce_mean( diff )
tf.summary.scalar( 'cross_entropy', cross_entropy )
with tf.name_scope( 'train' ):
  train_step = tf.train.AdamOptimizer( FLAGS.learning_rate ).minimize(
cross_entropy )
with tf.name_scope( 'accuracy' ):
  with tf.name_scope( 'correct_prediction' ):
    correct_prediction = tf.equal( tf.argmax( y, 1 ), tf.argmax( y_, 1 ) )
  with tf.name_scope( 'accuracy' ):
    accuracy = tf.reduce_mean( tf.cast( correct_prediction, tf.float32 ) )
tf.summary.scalar( 'accuracy', accuracy )
# 合并所有 summaries 信息并将其写到/tmp/mnist_logs（默认）目录。
merged = tf.summary.merge_all( )
train_writer = tf.summary.FileWriter( FLAGS.summaries_dir + '/train', sess.graph )
test_writer = tf.summary.FileWriter( FLAGS.summaries_dir + '/test' )
tf.global_variables_initializer( ).run( )
```

在初始化 FileWriters 之后，我们还必须添加 summaries 到 FileWriters 中作为我们的训练和测试模型。

```
# 训练模型中同样写入 summaries。
# 每 10 步，度量一次测试数据精度，并写入测试 summaries 中。
# 对于其他的步数，在训练数据中运行 train_step，并加入到训练 summaries 中。
def feed_dict( train ):
  """自定义一个 TensorFlow feed_dict 函数：将数据映射到 Tensor 占位符中。"""
```

```
  if train or FLAGS.fake_data:
    xs, ys = mnist.train.next_batch( 100, fake_data = FLAGS.fake_data )
    k = FLAGS.dropout
  else:
    xs, ys = mnist.test.images, mnist.test.labels
    k = 1.0
  return { x: xs, y_: ys, keep_prob: k }
for i in range( FLAGS.max_steps ):
  if i % 10 == 0:  # 记录 summaries 以及测试精度。
    summary, acc = sess.run( [ merged, accuracy ], feed_dict = feed_dict( False ) )
    test_writer.add_summary( summary, i )
    print( 'Accuracy at step %s: %s' % ( i, acc ) )
  else:  # 记录训练数据 summaries，然后训练。
    summary, _ = sess.run( [ merged, train_step ], feed_dict = feed_dict( True ) )
    train_writer.add_summary( summary, i )
```

- *启动* TensorBoard

接下来我们启动 TensorBoard 查看可视化的日志文件。首先，需要运行上述的 mnist_with_summaries.py 文件，生成定义好的日志文件。如图 8-15 所示，假设 TensorFlow 源码存放在：“F:\DLAction\TensorFlow-1.0\TensorFlow\examples\tutorials\mnist”目录下，只需要跳转到相应的目录，然后使用 Python 运行 mnist_with_summaries.py 文件即可。

```
F:\DLAction\tensorflow-1.0\tensorflow\examples\tutorials\mnist>python mnist_with_summaries.py
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\stream_executor\dso_lo
ader.cc:135] successfully opened CUDA library cublas64_80.dll locally
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\stream_executor\dso_lo
ader.cc:135] successfully opened CUDA library cudnn64_5.dll locally
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\stream_executor\dso_lo
ader.cc:135] successfully opened CUDA library cufft64_80.dll locally
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\stream_executor\dso_lo
ader.cc:135] successfully opened CUDA library nvcuda.dll locally
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\stream_executor\dso_lo
ader.cc:135] successfully opened CUDA library curand64_80.dll locally
Extracting /tmp/tensorflow/mnist/input_data\train-images-idx3-ubyte.gz
Extracting /tmp/tensorflow/mnist/input_data\train-labels-idx1-ubyte.gz
Extracting /tmp/tensorflow/mnist/input_data\t10k-images-idx3-ubyte.gz
Extracting /tmp/tensorflow/mnist/input_data\t10k-labels-idx1-ubyte.gz
```

图 8-15　运行 mnist_with_summaries 示意图

当运行训练完 mnist_with_summaries 后，接下来我们启动 TensorBoard，由于我们的 mnist_with_summaries.py 文件存放在 F 盘，默认情况下，TensorFlow 的日志文件会被写入到：“F:\tmp\TensorFlow\mnist\logs\mnist_with_summaries”目录，如图 8-16 所示，我们可以使用以下命令启动 TensorBoard。

```
tensorboard --logdir=F:\tmp\TensorFlow\mnist\logs\mnist_with_summaries
```

```
F:\DLAction\tensorflow-1.0\tensorflow\examples\tutorials\mnist>tensorboard --logdir=F:\tmp\tensorf
low\mnist\logs\mnist_with_summaries
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\stream_executor\dso_lo
ader.cc:135] successfully opened CUDA library cublas64_80.dll locally
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\stream_executor\dso_lo
ader.cc:135] successfully opened CUDA library cudnn64_5.dll locally
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\stream_executor\dso_lo
ader.cc:135] successfully opened CUDA library cufft64_80.dll locally
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\stream_executor\dso_lo
ader.cc:135] successfully opened CUDA library nvcuda.dll locally
I c:\tf_jenkins\home\workspace\release-win\device\gpu\os\windows\tensorflow\stream_executor\dso_lo
ader.cc:135] successfully opened CUDA library curand64_80.dll locally
Starting TensorBoard b'41' on port 6006
(You can navigate to http://113.55.54.233:6006)
```

图 8-16　TensorBoard 启动示意图

启动 TensorBoard 后，可以使用该网址：http://113.55.54.233:6006，通过浏览器可视化查看 TensorBoard。

8.5.2　计算图可视化

如图 8-17 所示，TensorFlow 计算图是非常强大的，但同时也非常复杂。在本小节中，我们将详细地介绍 TensorBoard 的计算图可视化，帮助你理解和调试。

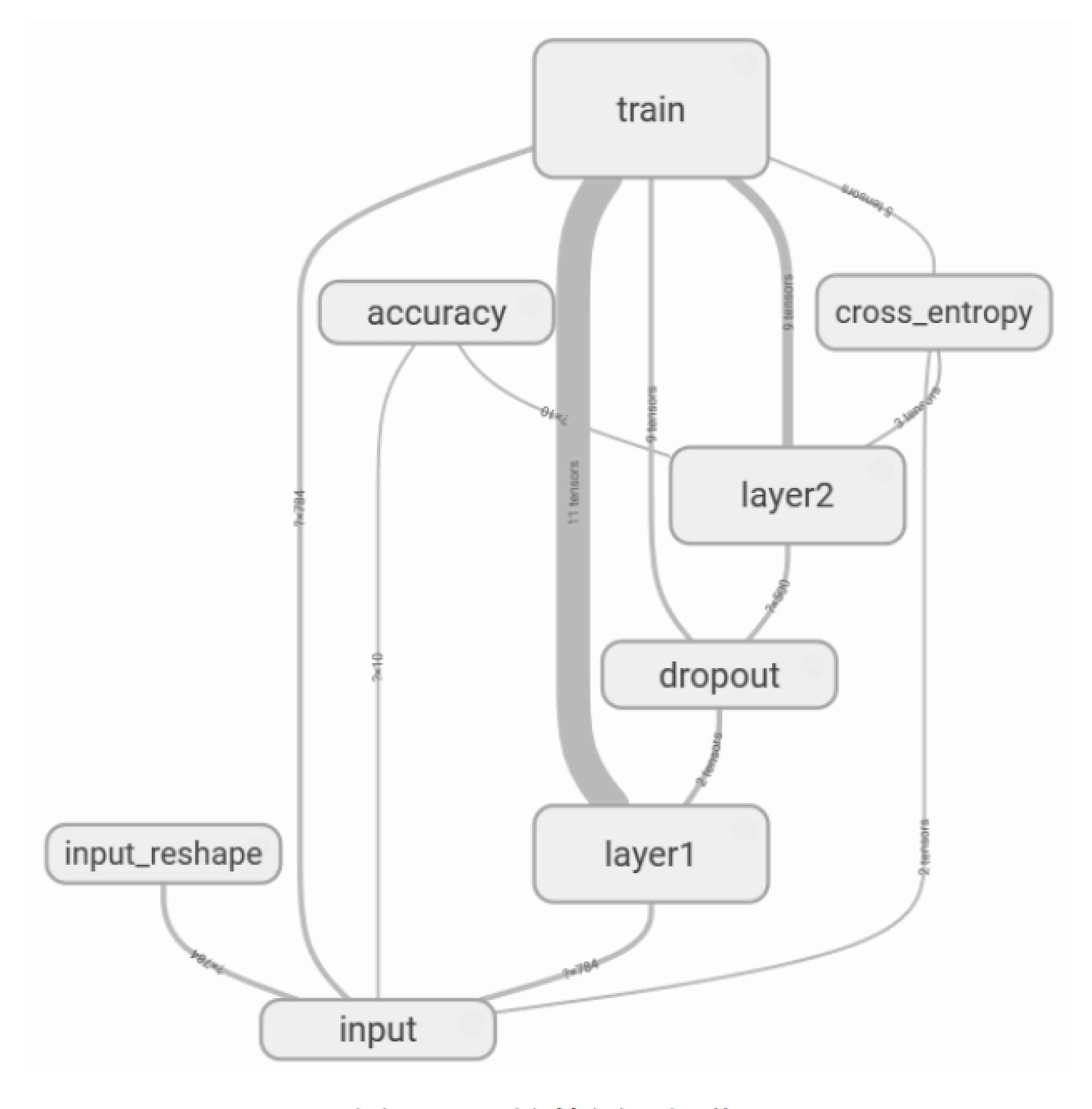


图 8-17　计算图可视化

- *名称作用域与节点*

典型的 TensorFlow 计算图可能拥有数千节点，即使拥有计算图工具，也无法同时轻松地查看所有节点。为了简化，变量名称可以被指定作用范围，然后使用其在计算图中定义层次结构的节点。在默认情况下，只有处在作用域最顶层的节点会被显示。如下列代码所示，我们使用 tf.name_scope 在“hidden”的名称作用域(name scope)下定义了三个操作。

```
import TensorFlow as tf
with tf.name_scope( 'hidden' ) as scope:
  a = tf.constant( 5, name = 'alpha' )
  W = tf.Variable( tf.random_uniform( [ 1, 2 ], -1.0, 1.0 ), name = 'weights' )
  b = tf.Variable( tf.zeros( [ 1 ] ), name = 'biases' )
```

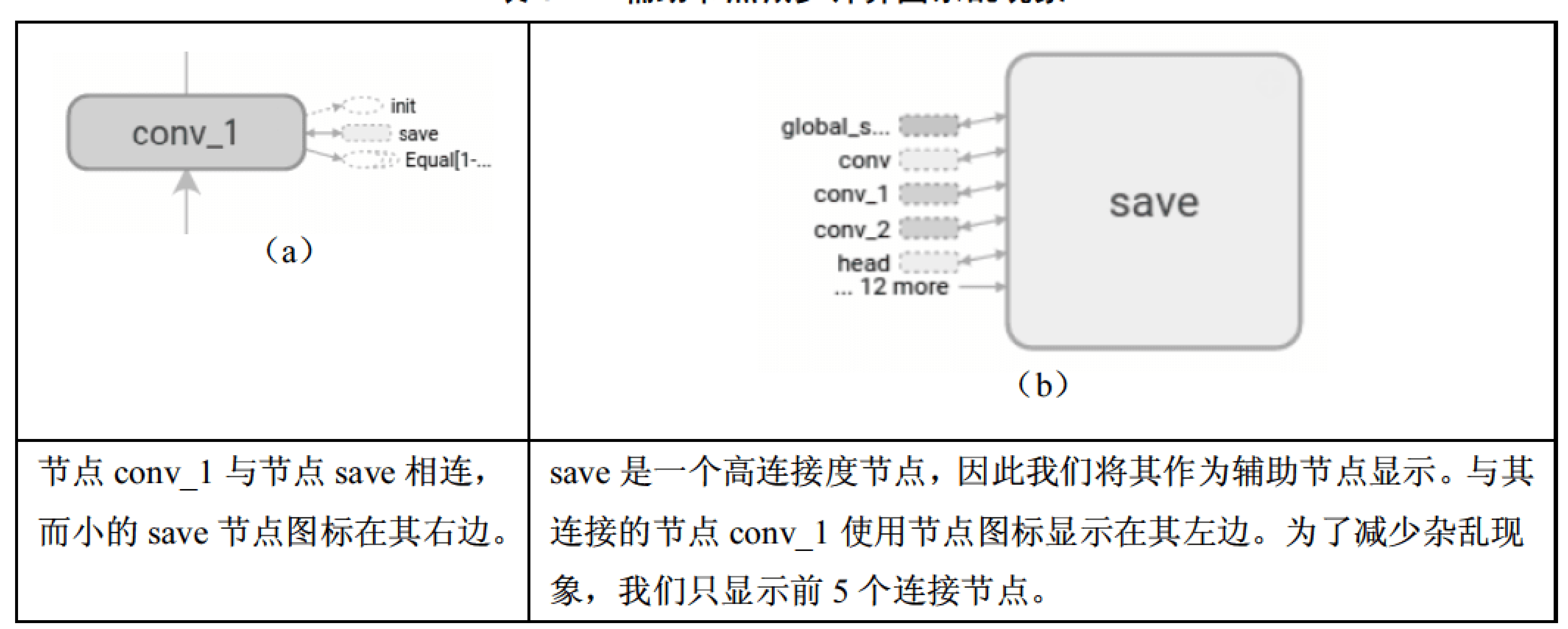
该操作形成了三个作用域：hidden/alpha、hidden/weights 和 hidden/biases。在默认情况下，这三个节点会被隐藏进一个标记为 hidden 的节点中。但详细的信息并没有丢失，可以双击或者单击节点右上角的橘黄色“+”，然后可以看见被隐藏的三个子节点：alpha、weights 及 biases。通过名称作用域组织节点，我们的计算图将变得非常清晰。当你构建模型时，注意将各个模块使用名称作用域分隔开。如表 8-1 所示为一个真实的作用域例子，表 8-1 中的（a）初始时看见的顶层名称作用域为 pool_1，单击橘黄色“+”按钮后，将显式子作用域节点，如表 8-1 中的（b）所示，为 pool_1 名称作用域扩展后的视图。再次单击橘黄色“-”按钮，将缩回原来的 poo_1 视图。

表 8-1　名称作用域，数据依赖，控制依赖说明

pool_1 （a）	pool_1 (pool_1)　Histogra...　Equal[1-... control_dependency CheckNumerics （b）
初始时看见的顶层名称作用域 pool_1。单击橘黄色“+”按钮后，其将为显式子作用域节点。	pool_1 名称作用域扩展后的视图。单击橘黄色“-”按钮将缩回原来的 poo_1 视图。

TensorFlow 计算图有两种连接方式：**数据依赖**（Data Dependencies）和**控制依赖**（Control Dependencies）。数据依赖显示了两个操作之间的 Tensor 流向，用实线箭头表示，而控制依赖使用虚线表示。表 8-1 中(b)图所示的扩展视图中，除了 CheckNumerics 和 control_dependency 的连接是虚线外，其他的连接都是用实线箭头连接的数据依赖。

表 8-2　辅助节点减少计算图杂乱现象



(a)	(b)
节点 conv_1 与节点 save 相连，而小的 save 节点图标在其右边。	save 是一个高连接度节点，因此我们将其作为辅助节点显示。与其连接的节点 conv_1 使用节点图标显示在其左边。为了减少杂乱现象，我们只显示前 5 个连接节点。

大多数 TensorFlow 计算图都有少数的节点被其他许多节点连接着。例如，许多节点都会在 init 节点上有一个控制依赖，绘制 init 节点和其依赖的所有边可能会使得视图非常散乱。为了降低这种散乱情况，如表 8-2 所示，我们可以将这种高连接度节点，分隔到右边的辅助区域（Auxiliary Area）并隐藏它们的边，我们绘制小节点图标代替线去表示其连接。一般情况下，被我们分隔除去的辅助节点都不会带走关键信息。为了帮助你理解 TensorFlow 中的图标，在表 8-3 中，我们总结了一些计算图中图标的含义。

表 8-3　计算图图标含义说明

图标	含义
	高层名称作用域的节点，双击后进行扩展
	没有彼此连接的序列节点
	彼此连接的序列节点
	单独的操作节点
	常数
	summary 节点
	操作节点间的数据流边
	操作节点间的控制依赖边
	双向箭头表示向外流的操作节点可以转变为向内流的 tensor

- Tensor 形状信息

序列化的 GraphDef 包含了 Tensor 形状，用 Tensor 维度标记边及用边的厚度表示 TensorFlow 的尺寸。为了在 GraphDef 中包含 Tensor 形状，当序列化计算图时，GraphDef 会传递真实的计算图对象到 FileWriter。如图 8-18 所示，为使用 Tensor 形状信息的 CIFAR-10 模型。

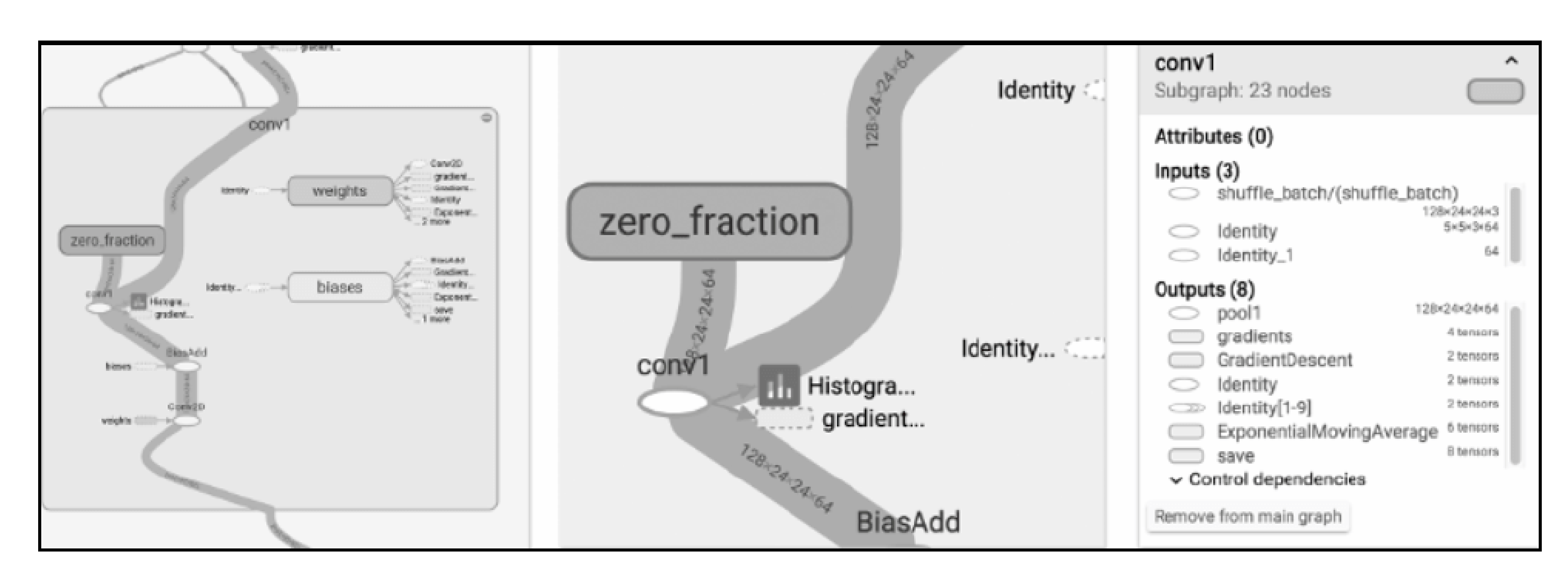


图 8-18　使用 tensor 形状信息的 CIFAR-10 模型

如图 8-19 所示，当启动 TensorBoard 并进入图形栏时，你将看见“Session runs”选项，其对应于每一步运行时被加入的**元数据**（metadata）。选择其中一个运行将会显示该步数下的网络快照，并将没有使用的节点屏蔽。在左边的控制面板中，还可以使用颜色标记节点的内存，计算时间等消耗情况，并且单击相应的节点将会显示精确的内存、计算时间与 Tensor 输出大小等信息。

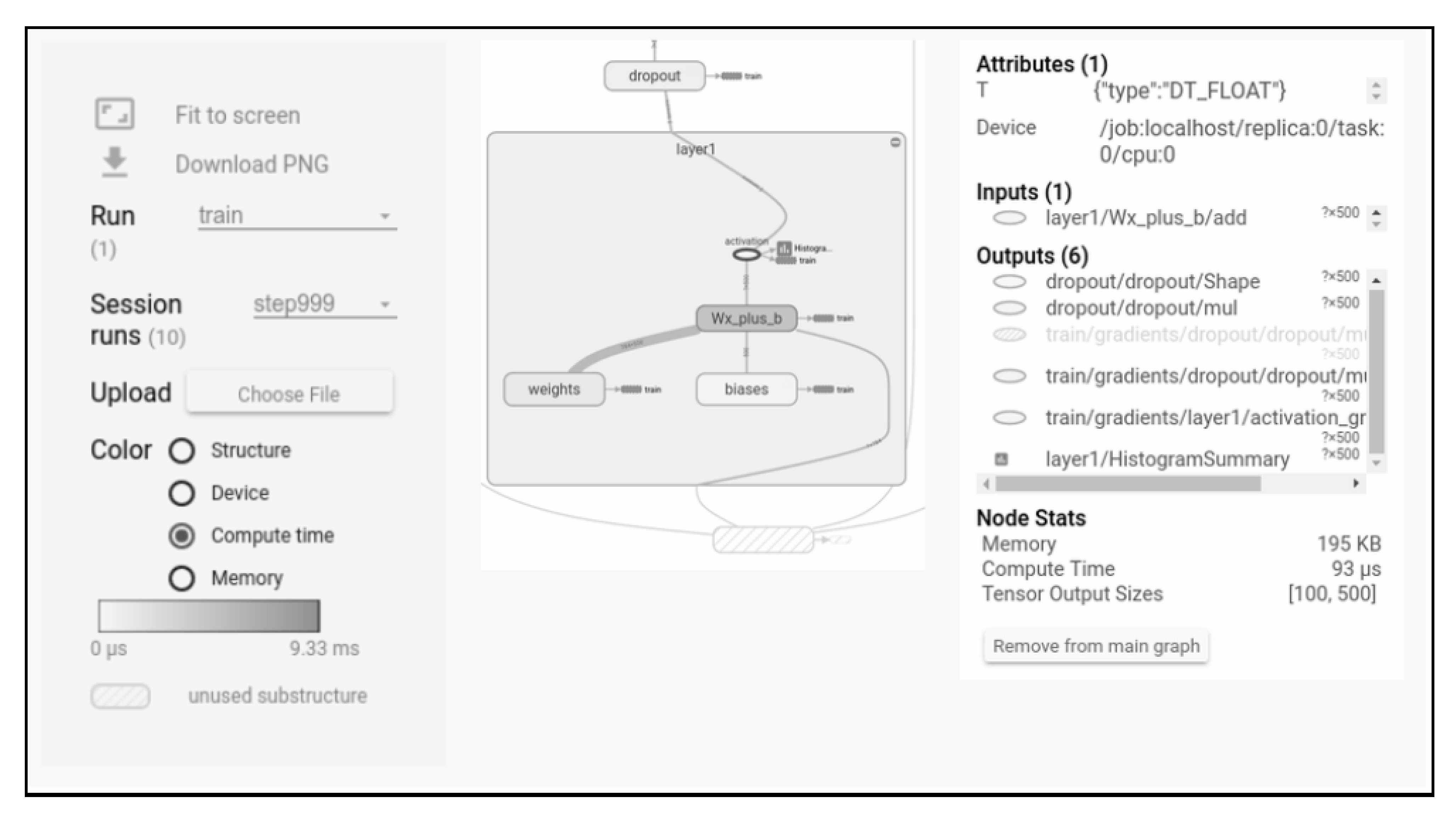


图 8-19　运行元数据计算图

以上内容只是 TensorFlow 的冰山一角，更多丰富的资源读者可以通过访问 TensorFlow 官方网站 http://www.TensorFlow.org/进行学习。

We can only see a short distance ahead，but we can see plenty there that needs to be done.

——Alan Mathison Turing





Python+ Spark 2.0+Hadoop

- 浅显易懂的原理介绍
- Step by Step 实机操作、范例程序详细解说
- 降低机器学习与大数据技术的学习门槛

内容简介

本书从浅显易懂的“大数据和机器学习”原理说明入手，讲述大数据和机器学习的基本概念，如分类、分析、训练、建模、预测、机器学习（推荐引擎）、机器学习（二元分类）、机器学习（多元分类）、机器学习（回归分析）和数据可视化应用等。书中不仅加入了新近的大数据技术，还丰富了“机器学习”内容。

为降低读者学习大数据技术的门槛，书中提供了丰富的上机实践操作和范例程序详解，展示了如何在单机Windows系统上通过Virtual Box虚拟机安装多机Linux虚拟机，如何建立Hadoop集群，再建立Spark开发环境。书中介绍搭建的上机实践平台并不限制于单台实体计算机。对于有条件的公司和学校，参照书中介绍的搭建过程，同样可以实现将自己的平台搭建在多台实体计算机上，以便更加接近于大数据和机器学习真实的运行环境。

本书非常适合于学习大数据基础知识的初学者阅读，更适合正在学习大数据理论和技术的人员作为上机实践用的教材。